Björn Eiderbäck Per Hägglund Olle Bälter

Objektorienterad programmering i Smalltalk

Studentlitteratur

Inlagan är tryckt på MultiFine Offset, licens nr 304 054

ISBN 91-44-49591-9

Omslagslayout: Kjeld Brandt

Printed in Sweden
Studentlitteratur, Lund

Tryckning:	1	2	3	4	5	6	7	8	9	10	1999	98	97	96	95

Innehåll

Förord

Vi har sedan 1985 undervisat i objektorientering och Smalltalk i ett otal kurser. Denna bok är en sammanfattning av svar på frågor som elever under årens lopp ställt till oss. Eleverna har befunnit sig på olika nivåer, allt från nybörjare till mycket duktiga programmerare. Eleverna vi släpper ifrån oss brukar ha en god uppfattning om objekt-orientering och Smalltalk och förhoppningsvis får även läsarna av denna bok det.

Vi kan inte svära oss fria från att det kan finnas fel i boken, även om all kod är testad och därefter kopierad och inklistrad. Alla fel som påpekas för oss för vi in på en world-wide-web sida hos Studentlitteratur, se http://www.studli.se/publishing/publishing.html.

Ett varmt tack till våra familjer och handledare som har visat ett oändligt tålamod med oss under vår frånvaro från våra huvudsakliga uppgifter: att umgås med familjen och doktorera.

Lättaste sättet att nå oss för frågor, klagomål och beröm är via datorpost:

bjorne | perh | balter @nada.kth.se

Välkommen till en spännande värld med objekt, instanser och arv utan tjafs med obegriplig syntax!

Stockholm i juni 1995,

Björn, Per och Olle

1 Grunder

Detta kapitel besvarar bland annat följande frågor:

- Hur uppstod objektorientering?
- Vad är skillnaden mellan strukturerad programmering och objektorienterad programmering?
- Vilka är de grundläggande termerna vid objektorientering?
- När har man störst fördelar av objektorienterad programmering?
- Vilka är Smalltalks fördelar och nackdelar?
- Hur använder man Smalltalks utvecklingsmiljö?
- Hur skriver man i Smalltalk språkkonstruktioner som val, iteration och anrop till metoder?

1.1 Inledning

Alla som har erfarenheter av att utveckla ett större programsystem har också erfarenheter av problemen: Ständiga ändringar av ”låsta” specifikationer, programmoduler som är beroende av andra moduler och inte går att ändra utan omfattande arbete på alla andra moduler också. Objektorientering är *ett* bra sätt att handskas med de problem som uppstår när man konstruerar större system.

Objektorientering är en *metodik* och innebär ett nytt sätt att tänka vid programutveckling. Man utgår ifrån de objekt som programmet ska kunna hantera och lägger därefter till de operationer man vill kunna utföra på objekten. Varje objekt och dess operationer konstrueras var för sig, utan att vara beroende av hur andra objekt ser ut ”inuti”.

Fördelen med detta är att objekt som finns i verkligheten normalt är mer långlivade än tex de signaler som går in och ut ur ett system. Ett system som är systemerat efter signaler kommer att bli mycket mer känsligt för ändringar och dessutom blir det svårt att återanvända kod.

Förutsättningar

Läsaren av denna bok antas ha erfarenheter av programmering. Begrepp som variabel, parameter, program, tilldelning, sekvens, iteration, villkor, kompilator, strukturerad programmering, avlusare och fil förutsätts vara välkända.

Objektorienteringens fördelar märks tydligast i stora system. Att läsa en bok med endast små programsnuttar ger dålig förståelse för fördelarna med objektorientering. Vi har därför valt att använda oss av mindre exempel för att introducera begrepp och lite större återkommande exempel för att ge läsaren en känsla för kraftfullheten objektorientering ger.

Mål

Efter att ha tillägnat sig innehållet i denna bok kommer läsaren att veta vad objektorientering är, kunna skriva program i Smalltalk. Kunskaperna kommer att vara tillräckligt djupa för att du ska kunna tillgodogöra dig boken om interaktionsprogrammering, se referenslistan på sidan 455

Historik

Under slutet av 50-talet blev behovet av programspråk som underlättade strukturering av program allt mer uppenbar och resulterade runt 1960 i bland annat Algol, Cobol och Lisp.

På Algol baserades sedan ett stort antal språk som tex Simula, Pascal och Modula2. Några visionärer, Ole-Johan Dahl och Kristen Nygaard, insåg i mitten av sextiotalet att strukturerad programmering var nödvändigt, men inte tillräckligt och introducerade *klassbegreppet* i Simula-67. Klassbegreppet är centralt för all objektorientering. En klass är en beskrivning av ett objekt med såväl data som operationer man kan göra på dessa data.

På sextiotalet var simulering det mest avancerade man kunde göra med en dator och Simula konstruerades för att enkelt kunna utföra avancerade simuleringar. Simula blev mycket omtyckt i den akademiska världen i Norden och på några ställen i USA, men slog aldrig i industrin. Världen var ännu inte mogen för objektorientering.

Nya programspråk kommer och går ideligen, men under åttiotalet lanserades två nya objektorienterade språk som har fått stor spridning både i industrin och i den akademiska världen: Smalltalk-80 ursprung-

ligen från Xerox Parc och C++ som skapades av Bjarne Stroustrup på Bell Laboratories. Bägge baseras på klassbegreppet i Simula.

Smalltalk föddes i början på sjuttiotalet från en vision av Alan Kay vid Xerox PARC (Palo Alto Research Centre). Målet var att skapa ett lättanvänt verktyg för barn och resulterade i Smalltalk-72, implementerat med 3000 rader Basickod.

Grundläggande begrepp i Smalltalk-72 var objekt, klasser och metoder. Ett annat Xerox PARC-projekt vid den här tiden var en grafisk arbetsstation som i form av Alto (1976) hade musstöd, högupplösande grafisk skärm, fönster, ikoner, menyer, sköldpaddsgrafik (turtle graphics, inspirerat av LOGO på MIT), direktmanipulation, (inspirerat av Sketchpad av Sutherland och FLEX av Kay), bitkarte-baserad text mm. För Alto utvecklades Smalltalk till en avancerad programmeringsmiljö med möjligheter till att konstruera interaktiva grafiska tillämpningar för arbetsstationer. I Smalltalk-76 myntades själva begreppet objektorientering.

Flera av utvecklarna på PARC gick över till Apple och tog med sig sina erfarenheter av den interaktiva grafiska miljön vilket bidrog till den grafiska gränssnittsutvecklingen för Apples Macintosh, lanserad 1984, och den grafiska revolutionen i datorvärlden.

Den grafiska interaktionsmiljön är mycket svår att bemästra även med strukturerad programmering, varför det objektorienterade tankesättet kom att slå igenom stort. Grafisk interaktionsprogrammering utan objektorientering är idag (nästan) otänkbart, men objektorientering är även användbart till mycket annat, särskilt när systemen blir stora.

1.2 Varför objektorientering?

Med traditionella programspråk uppstår det problem när programmen når upp mot i storleksordningen 10.000 rader kod. Varje fel man hittar och rättar orsakar då ofta ett annat fel på ett helt annat ställe. De objektorienterade språken utvecklades för att man skulle klara av ännu större programsystem bla genom att återanvända kod och kapsla in programdelar och därmed göra dem mindre beroende av varandra.

Förutom problemen med stora program märker man ofta som programmerare hur vissa problem ständigt återkommer: sökning, sortering, filhantering. Varje gång krävs det en lösning som liknar alla de tidigare lösningarna, men den måste modifieras i något avseende som ofta kräver mycket programmerande och testande.

Programbibliotek

Idag bygger ingen programsystem utan att använda biblioteksmoduler. Det vore lika dumt som att konstruera komplicerade kretskort utan integrerade kretsar, bilar med handgjorda skruvar eller bygga ett matsalsbord genom att köpa skog, fälla ett träd, torka det, såga upp trädet, hyvla plankorna och sätta ihop dem med bultar som man har smitt av järn ur den egna gruvan.

Klassbibliotek

Objektorienterade system medför att man får standardiserade delar. Detta är ingenting nytt, det får man med alla biblioteksmoduler. Skillnaden är att de objektorienterade biblioteksmodulerna är enklare att anpassa till nya problemområden.

I programbibliotek finns det vanligen en uppsättning procedurer och funktioner som endast kan anropas på ett visst sätt och lämna tillbaka svar på ett sätt. Med ett klassbibliotek får man istället en uppsättning generella datatyper och metoder. Dessutom kan man sedan om det behövs själv anpassa dem till det aktuella problemet.

Som exempel tar vi sortering. För att sortera små mängder, upp till 50 element, är insättningssortering enkel och snabb. För större mängder är quicksort en bra metod. Vad man vill sortera varierar, men för att göra exemplet hanterbart begränsar vi oss till heltal, flyttal och strängar.

En traditionell biblioteksmodul skriven i Fortran eller något annat föråldrat språk skulle då behöva innehålla sex subrutiner:

Subrutin	Metod	Typ
`insint`	insättning	heltal
`insreal`	insättning	flyttal
`insstr`	insättning	strängar
`quickint`	quicksort	heltal
`quickreal`	quicksort	real
`quickstr`	quicksort	strängar
Vektorn som skall sorteras och antal elementens blir parametrar till subrutinerna		

Figur 1.1 Subrutiner i Fortran

Ett anrop kan se ut så här:

```
        IF ((vecsize.LE.50) .AND. inttype) THEN
          CALL insint(vec, vecsize)
          GOTO 100
        ENDIF
        IF ((vecsize.LE.50) .AND. realtype) THEN
          CALL insreal(vec, vecsize)
          GOTO 100
        ENDIF
        IF ((vecsize.LE.50) .AND. strtype) THEN
          CALL insstr(vec, vecsize)
          GOTO 100
        ENDIF
        IF ((vecsize.GT.50) .AND. inttype) THEN
          CALL quickint(vec, vecsize)
          GOTO 100
        ENDIF
        IF ((vecsize.GT.50) .AND. realtype) THEN
          CALL quickreal(vec, vecsize)
          GOTO 100
        ENDIF
        IF ((vecsize.GT.50) .AND. strtype) THEN
          CALL quickstr(vec, vecsize)
          GOTO 100
        ENDIF
  100   CONTINUE
```

De logiska variablerna **inttype**, **realtype** och **strtype** anger typen för vektorns (**vec)** element. I modernare programspråk som Pascal och C kan man skicka funktioner som parametrar och klarar sig därmed med två procedurer från biblioteket. Däremot måste man skriva tre funktioner som anger hur två element är relaterade till varandra enl figur 1.2.

Funktionshuvud

`void insertionsort(vectype *v, int n, int(* less)())`
 	
`void quicksort(vectype *v, int n, int(* less)())`

v är vektorn som ska sorteras, **n** är antalet element, **vectype** är definierad som **typedef elementtype vectype[]** och funktionen **less** ska returnera true om **a < b**.

Figur 1.2 Biblioteksfunktioner i C

Om vi vill sorterera heltal skriver vi funktionen **lessInt**. Den anger om första parametern är mindre än den andra på det här sättet:

```
int lessInt(elementtype a,b)
/* returnerar true om a < b */
{
   return(a<b);
}
```

Ett anrop kan se ut så här:

```
if (vecsize<=50)
  insertionsort(vec, vecsize, lessInt);
else
  quicksort(vec, vecsize, lessInt);
```

Vill man byta typ måste man definiera om **elementtype** och byta ut **lessInt** mot **lessString**:

```
int lessString(elementtype a,b)
/* returnerar true om a < b */
{
   return(strcmp(a,b)<0);
}
```

Men typberoendet i **vec** och **less** går inte att komma ifrån på något enkelt sätt. Vill man hantera flera typer samtidigt måste man definiera ytterligare funktioner som kan flytta på olika typer av data mellan två fack i en vektor. Koden ovan kan endast sortera element av en enda typ. Det man vinner på att använda C eller Pascal istället för Fortran är att antalet biblioteksfunktioner blir färre. Det behövs inte en funktion för varje datatyp utan en för varje sorteringsmetod *om man kan stå ut med begränsningen att man endast kan sortera med avseende på en datatyp per program!*

Objektorienterad sortering

Det objektorienterade sättet att lösa problemen med sorteringen ovan är att definiera en less för varje typ. Därefter går anropet (Smalltalksyntax)

```
vec sort
```

att utföra oavsett vilken typ elementen i vec har. Vill man sortera heltalen i intvec och strängarna i stringvec kan man i samma program skriva

```
intvec sort.
stringvec sort.
```

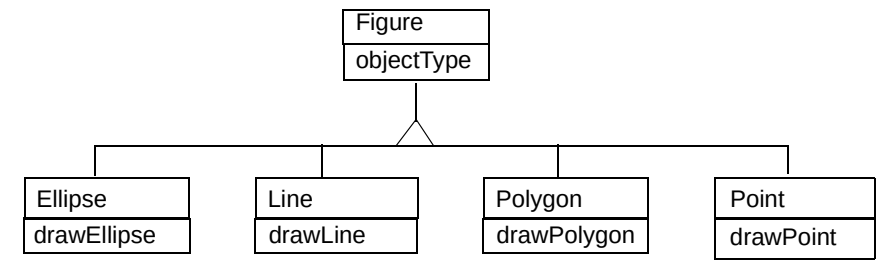
Detta går inte i C-programmet (eller i någon standardpascalversion), eftersom det bara får finnas *ett* **less** i programmet! Den objektoriente-

rade lösningen bygger på att *ansvaret* för hur två element ska relateras till varandra *delegeras* till den klass som beskriver elementtypen och att olika metoder (procedurer, funktioner och subrutiner) får ha samma namn (polymorfi).

Det blir ofta lika mycket kod att skriva, men den objektorienterade lösningen blir både flexiblare och robustare. Ett tillägg med ytterligare datatyper förstör inte den kod som fungerade med de gamla typerna. Tilläggen som görs blir endast de nya datatyperna, ingenting behöver ändras i koden som anropar sorteringsmetoderna!

Objekthierarki

Verkliga objekt ”vet” hur de ska se ut och vilka egenskaper de har, men i ett program är programmeraren Kung och bestämmer allt. Skriver programmeraren att en cirkel ska ritas som en kvadrat så blir det så

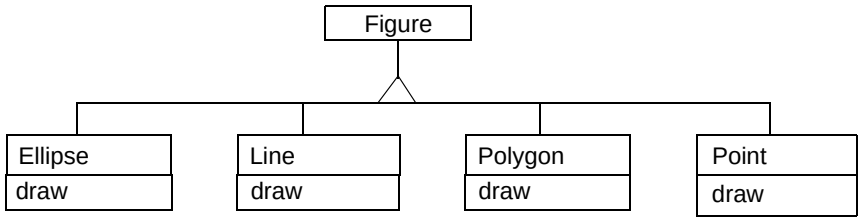


Figur 1.3 Hierarki av record i ett icke objektorienterat språk

I ett program för att rita ut enkla grafiska objekt på skärmen kan det se ut på följande sätt (Pascalsyntax):

```
CASE figure.objectType OF
          Ellipse  : drawEllipse(figure);
          Line     : drawLine(figure):
          Polygon  : drawPolygon(figure);
          Point    : drawPoint(figure);
END; {CASE}
```

Beroende av typen på **`figure`** kommer olika procedurer för att utföra utritningen att anropas.



Figur 1.4 Objekthierarki med utritningsmetod

Det objektorienterade sättet är (Smalltalksyntax):

```
figure draw.
```

Fördelen med det objektorienterade sättet är inte främst att koden blir mycket kortare utan att *ansvaret* för hur objekt ska ritas ut ligger hos *objekten själva*, inte i någon ritmodul. Lägger man till en objekttyp så måste man i det första, strukturerade fallet, lägga till en rad i varje kommando (draw, erase, ...). I det sista objektorienterade fallet behöver ingenting ändras! Det enda man behöver tänka på är att det nya objektet måste förstå draw, erase, osv. Objekten själva har ansvaret för sitt beteende och utseende, precis som verkliga objekt!

Programutveckling

Alla utvecklingsmetoder, där man låser en specifikation och därefter konstruerar systemet, bygger på att man vet allt som kan vara värt att veta om systemet redan innan det är konstruerat. Det kan jämföras med att lösa en differentialekvation med Eulers metod genom att använda ett enda steg, förutsäga vädret nästa vecka baserat på dagens väder, eller att titta på pilotavsnittet i en TV-serie som Dallas eller Rederiet och försöka avgöra vem som kommer att göra vad med vem i det sista avsnittet.

Systemet med ”låsta” specifikationer (sk vattenfallsmetodik där varje fas av utvecklingen avslutas innan nästa påbörjas) utnyttjar inte att människorna som bygger systemet lär sig mer och mer om problemområdet och kommer med innovationer under utvecklingsarbetet. Detta leder till att man inte utnyttjar utvecklarnas hela kapacitet och utvecklingsarbetet blir mekaniskt. Även användarna stängs ute från att ha några synpunkter under utvecklingstiden och tvingas till att acceptera eller förkasta hela systemet när det är klart.

Objektorienterad programutveckling trollar inte bort specifikationsändringarna, men gör det lättare att ta hand om dem eftersom objekten i systemet är relativt oberoende av varandra.

Decentraliseringen gör att tillägg endast blir tillägg och inte omkonstruktion. Ändringar blir ändringar endast på de ställen som ska ändras, obehagliga bieffekter uteblir[1].

1.3 Terminologi

Eftersom objektorientering är en metodik som skiljer sig från traditionell programmering finns det också nya termer som beskriver nya begrepp. En del nya termer beskriver gamla begrepp men då ofta mer generellt. Här följer en genomgång av de viktigaste begreppen.

Objekt, metoder, attribut och klasser

I objektorienterad analys (OOA) startar man med att identifiera de objekt som man vill hantera. Objekten kan vara objekt ur verkligheten som tex en båt, en person eller ett register. Objekten kan också vara mer abstrakta som en fil eller ett skärmfönster. I Smalltalk är allt objekt, även tal och textsträngar. Ett objekt har olika egenskaper som delas in i funktionalitet och attribut. Funktionaliteten beskrivs av olika *metoder*, (procedurer och funktioner). Attribut är värden som hör till ett objekt. På en textsträng kan man använda tex metoderna sammanslagning (konkatenering), utskrift och kopiering, medan dess längd är ett attribut. I Smalltalk beskrivs objekt och dess attribut i en *klassdefinition*.

Instansiering och skräpsamling

En *klass* är endast en beskrivning av hur ett objekt skulle se ut och fungera om det existerade, ungefär som en TYPE-definition i Pascal eller en typdefinition i C. När man verkligen vill ha ett objekt skapar man en *instans* av den aktuella klassen (instansierar). Jämför med **`New`** i Pascal och **`malloc`** i C. När en instans inte används längre tas den bort automatiskt av skräpsamlaren (Eng: garbage collector) i Smalltalk och Simula. I C++ måste programmeraren själv ta bort den, precis som i Pascal och C med **`Dispose`** respektive **`free`**.

---

[1] Om systemeringen är vettigt gjord. Det går att producera skräp med alla metodiker.

Polymorfi

Polymorfi innebär att olika metoder kan ha samma namn genom att de är definierade för olika objekt. Exempel:

Antag att vi har en klass String för textsträngar och en klass Vector för vektorer. Längden av en sträng är antalet tecken. Längden av en vektor är den euklidiska längden, dvs roten ur summan av kvadraten på alla komponenter. Om s är en sträng och v är en vektor kan man skriva: s length resp v length i Smalltalk och `s.length()` resp `v.length()` i C++. Typen på det objekt s resp v refererar till avgör hur längden beräknas. Det går inte att skriva både `length(s)` och `length(v)` i varken Pascal eller C pga typkontrollen eftersom length endast kan ta argument av en typ.

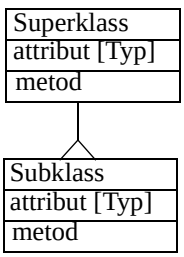
Dynamisk bindning

Vilken metod som används avgörs av klasstillhörigheten (v är en Vector och s är en String) och i Smalltalk dessutom vid exekveringen och inte vid kompileringen. Detta kallas för *sen* eller *dynamisk* bindning, i motsats till *tidig* eller *statisk* bindning.

Arv

Arv i programmeringssammanhang innebär att egenskaper (metoder och attribut) ärvs i en hierarkisk ordning.

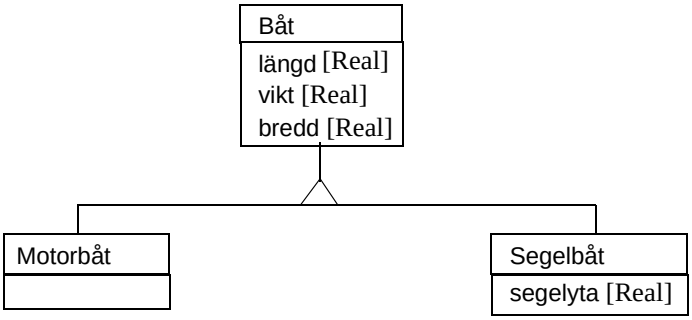
I denna bok använder vi figurer av nedanstående typ då vi illustrerar klasser och deras egenskaper:



Figur 1.5 Klassdiagram: Exempel, ointressanta delar kan utelämnas

Exempel: Båt med subklasser

Utgående från klassen Båt, kan man definerera subklasser Motorbåt och Segelbåt i figur 1.6. Klassen Båt kommer då att vara en *superklass* till



Figur 1.6 Klassdiagram: Motorbåt och Segelbåt ärver från Båt

Motorbåt och Segelbåt. I klassen Båt definieras bla egenskapen längd. Denna egenskap ärvs av Motorbåt och Segelbåt som alltså inte behöver definiera egenskaper som finns definierade i superklassen. Attribut kan i Smalltalk inte omdefinieras i en arvshierarki.

En lista bör ingå i alla klassbibliotek. En lista kan man fråga om bland annat: dess storlek, om den är tom eller om den innehåller ett visst element eller lägga till ett element till listan. I Smalltalk kan man använda klassen OrderedCollection och bland annat metoderna size, isEmpty, includes: respektive add: för detta.

Behöver man ett nytt objekt med liststruktur deklarerar man tex String som en *subklass* (härledd klass, arvtagare) till Collection (det är naturligtvis redan gjort i Smalltalk-systemet). Subklassen String ärver då automatiskt alla egenskaper som finns definerade för en Collection. Metoderna nämnda ovan är alla sådana som man behöver även för en String och de behöver alltså inte definieras igen.

Finns det metoder som inte passar kan man definiera om (eng: override) dem i subklassen. En omdefinierad metod har samma namn som metoden i superklassen, men ett annorlunda beteende. Omdefinieringen medför att man i objekt av subklassens typ använder subklassens metod och i objekt av superklasssens typ används fortfarande superklassens metod.

Fördelen med att ärva egenskaperna från listan är att man slipper skriva ny kod, eller upprepa gammal, för att ta hand om strängarnas listoperationer. Den välbeprövade koden för listan är förhoppningsvis felfri, så man slipper en massa felkontroller för textsträngarna.

Virtuella metoder

Antag att man i exemplet med båtarna har en instanst b ur klassen Båt, en instans m ur klassen Motorbåt och en instans s ur klassen Segelbåt. Då är följande metodanrop möjliga: b längd, b vikt, b bredd, m längd, m vikt, m bredd, s längd, s vikt, s bredd samt s segelyta. Däremot är följande metodanrop felaktiga: ~~b segelyta~~ och ~~m segelyta~~. Om man skulle ha en metod, foo i både Båt och Segelbåt så syftar b foo på metoden i Båt medan s foo syftar på metoden i Segelbåt. Detta är ett naturligt sätt att hantera namnkollisioner, men i andra objektorienterade språk är det inte självklart att det fungerar så. I tex Simula kan en pekare av typen Båt peka på ett objekt av typen Segelbåt och då måste man skilja på statiska och *virtuella* metoder. I Smalltalk blir alla metoder virtuella eftersom typen hos variabler avgörs av vilket objekt de pekar på för tillfället och inte vad de har deklarerats för.

Skillnad mot kopiering

Liknande saker går att åstadkomma om man tex kopierar koden för Båt till både Motorbåt och Segelbåt istället för att ärva från Båt, men det finns en viktig skillnad: När man ändrar i Båt måste man komma ihåg att göra om kopieringen till Segelbåt och Motorbåt. När Segelbåt istället ärver egenskaperna från Båt kommer alla ändringar i Båt att få genomslag i både Segelbåt och Motorbåt direkt. Arv är ett enkelt och kraftfullt sätt att administrera ändringar.

Abstrakta datatyper

För att få program stabila brukar man konstruera dem i moduler som är så oberoende av varandra som möjligt. Vid objektorientering kallas dessa moduler för klasser. Det finns liknande modularisering under namnet abstrakta datatyper, men en klass innehåller såväl en abstrakt datatyp som arv och polymorfi.

Det som är viktigt att definiera är själva gränssnittet, dvs vad datatypen ska heta, vilka metoder man ska kunna komma åt utifrån och vad metoderna ska returnera. Hur problemen löses internt för att utföra det som önskas lämnas åt den som ska implementera klassen. Det som är väsentligt för andra som ska använda sig av datatypen är nämligen: namnet på datatypen och dess metoder, parametrarna, samt vad metoderna returnerar.

1.4 Smalltalks interaktionsmiljö

Smalltalks utvecklingsmiljö är grafisk. Utvecklingsmiljön är i sig ett utmärkt exempel på objektorientering. På grund av att utvecklingsmiljön är integrerad med språket så är miljön en del av det som måste läras ut. För den som vill lära sig att behärska utvecklingsmiljön finns det utmärkta ”Tutorials”– introduktioner som följer med Smalltalk-systemen. Endast de mest grundläggande delarna gås igenom även här i boken. Bilder som visar hur programmeringsmiljön ser ut är från Parc-Places VisualWorks II på Sun under Motif eller Macintosh. Eftersom antalet musknappar kan variera använder vi beteckningen *selektions-*, *operations-* och *fönsterknapp* när vi beskriver interaktionsmiljön.

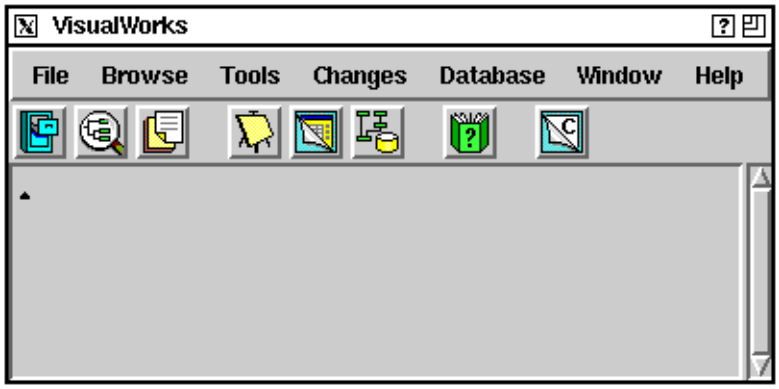
Knapp	**Sun**	**PC**	**Macintosh**
selektion	vänster	vänster	musknappen
operation	mitten	höger	alt+musknapp
fönster	höger	ctrl+höger	kommando+musknapp

Figur 1.7 Standardmusknappar på olika plattformar

Huvudmenyn-Launcher

I figur 1.8 nedan finns en bild av huvudmenyn i Smalltalksystemet. Fönstret består dels av en menyrad, dels av en rad med knappar för snabbkommandon och dels ett fönster för utmatning, utskriftsfönstret. (eng. Transcript). Via menyerna och knapparna kan man få fram alla andra delar av Smalltalkmiljön. Här beskrivs de tre viktigaste: Visual-

Works huvudfönster med utskriftsfönster (eng Transcript), arbetsfönster (eng Workspace) och System Browser.



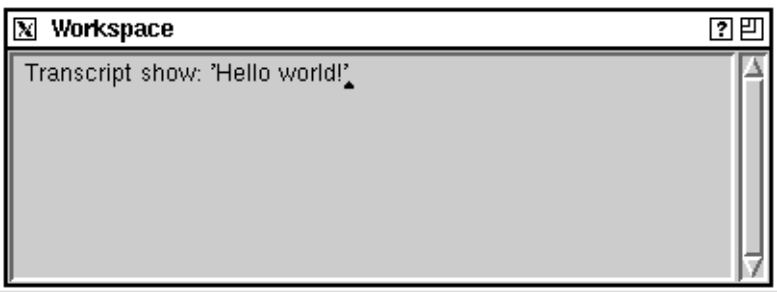
Figur 1.8 Huvudmenyn (Launcher) med tomt utskriftsfönster

Arbetsfönster-Workspace

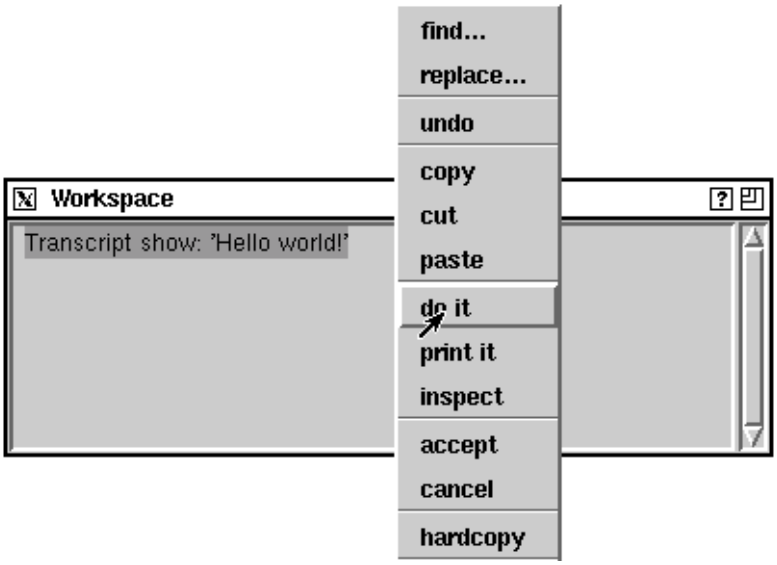
Fönstret med titeln ”Workspace” kan ses som ett skissblock där man kan testa Smalltalkuttryck. Det berömda ”Hello world!”-exemplet från Kernighan & Ritchies bok ”The C programming language” går att göra i Smalltalk genom att skriva

```
Transcript show: 'Hello world!'
```

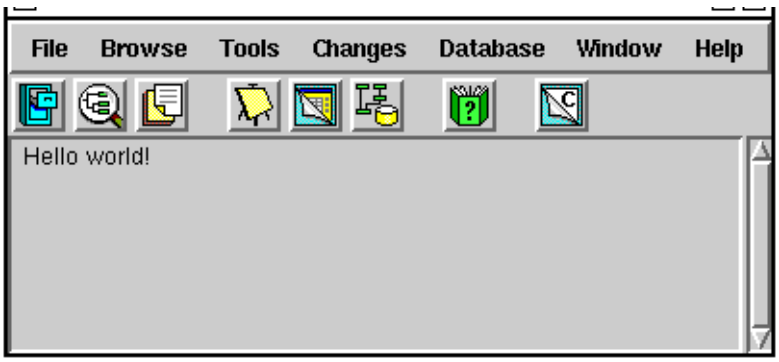
i arbetsfönstret, måla över texten med selektionsknappen nedtryckt och därefter välja ”do it” i operationsmenyn (menyn som dyker upp när man trycker på operationsknappen i ett fönster). Texten ”Hello World!” kommer då att skrivas ut i utskriftsfönstret.



Figur 1.9 Koden inskriven i arbetsfönster



Figur 1.10 Texten markeras och ”do-it” väljs i operationsmenyn



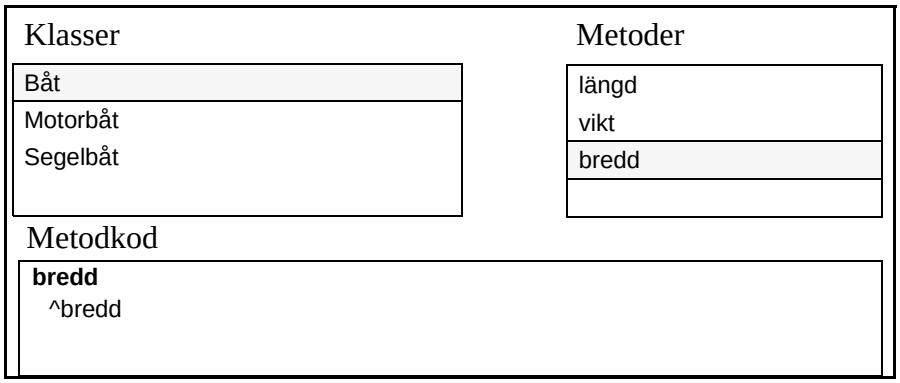
Figur 1.11 ”Hello world!” dyker upp i utskriftsfönstret

Kodkatalog-Browser

När man skriver stora program får man snabbt problem med överblick. De flesta redigeringsprogam (tex Emacs) kan endast visa koden på ett sätt, radindelad på samma sätt som filen lagras.

I Smalltalk kan man redigera kod i alla textfönster men det finns specialdesignade fönster av flera olika typer som visar koden på mer överskådligt sätt.

För små program klarar man sig med en uppdelning i klasser, metoder och kod, dvs ett fönster i tre delar. En första del där man ser vilka klasser som finns en andra som visar vilka metoder som finns och en tredje där koden visas. För att underlätta överblick får man endast se de metoder som finns i den klass som är vald och endast koden till den metod som är vald.



Figur 1.12 Fiktiv browser med endast klasser och metoder

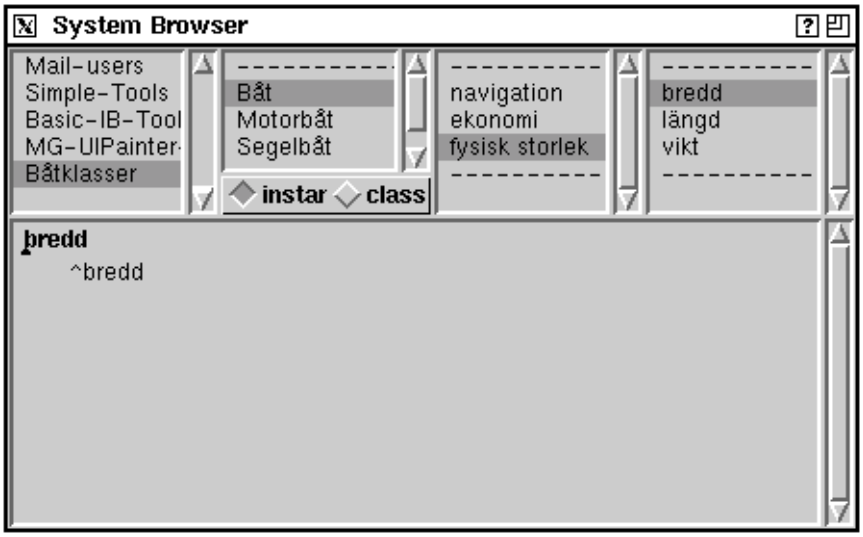
Det finns ytterligare två möjligheter som underlättar överbliken över koden, nämligen indelning av klasser i klasskategorier och metoder i metodkategorier.

1.5 Smalltalk

Många programmerare har en fanatisk inställning till det programspråk de själva använder. Trots detta anser även många icke-Smalltalkprogrammerare att Smalltalk är det bästa sättet att *lära* sig objektorientering.

Vår egen något fanatiska inställning är att:

- Smalltalk är det bästa språket vi har hittat för att få programmerare att förstå objektorientering.
- Smalltalk är en utmärkt miljö för att ta fram interaktiva tillämpningar.



Figur 1.13 System Browser i Visual Works

- Smalltalk är en utmärkt miljö att konstruera många typer av stora tillämpningar.
- För små, icke-interaktiva eller extrema realtidstillämpningar kan det finnas bättre alternativ.
- Den som tillgodogjort sig Smalltalk skriver mycket välstrukturerade program, oberoende av vilket språk de använder.

Smalltalk skiljer sig från andra språk genom att allt i språket beskrivs som objekt. Från heltal och textsträngar till browser och kompilator. Smalltalk innehåller klasser som hanterar grafik. Utvecklingsmiljön är integrerad med språket. En del av systemet består av en *virtuell maskin* (se nedan) som är datorberoende.

Syntaxen och semantiken är otroligt enkel: Sex reserverade ord (nil, true, false, self, super, thisContext), femton specialtecken ((|[]|{}_^;$#:.-()) och endast tre sätt att skicka meddelanden (unära, binära och nyckelord). Tack vare den virtuella maskinen och språkets objektorienterade uppbyggnad kan man köra Smalltalkprogram på såväl Sun, Macintosh som PC, utan ändringar i koden! System utvecklade med Smalltalk blir alltså flyttbara.

En ANSI-standard är under utveckling. I avvaktan på ANSI-Smalltalk använder vi i den här upplagan av boken ParcPlaces Smalltalk-80 som finns för alla plattformar. Värd att nämna är också några andra stora Smalltalkprodukter, Smalltalk\V för persondatorer från Digitalk. Smalltalk V följer fönsterstandarden på respektive plattform (och grafikdelarna blir därmed inte flyttbara). Smalltalk Agents från Quasar Knowledge Systems som för närvarande bara finns för Macintosh. VisualAge från IBM, för PC, SmalltalkX, HP's distribuerade implemetation av VisualWorks, Envy och GNU-Smalltalk.

Värt att nämna är att ParcPlace Systems och Digitalk har slagits samman och planerar att slå samman sina produkter och leverera en Smalltalk med godbitarna från både VisualWorks och Digitalk. Denna produkt går under arbetsnamnet ObjectMerger.

Virtuella maskinen

Smalltalk kompileras i två steg. Själva Smalltalksatserna kompileras och lagras för den virtuella maskinen, som bytekod, när satserna sparas[1] första gången. Första gången satserna utförs kompilerar den virtuella maskinen bytekoden till den aktuella maskinens objektkod och lagrar den i cache-minne för senare behov.

Fördelarna med detta sätt är flera:

- När man utför en ändring behöver man inte kompilera om hela systemet, utan kan köra programmet direkt. Detta är mycket praktiskt vid programkonstruktion och programutveckling; man ser direkt vad man har åstadkommit utan att behöva kompilera, länka, ladda och köra programmet.
- När man stöter på ett exekveringsfel kan man direkt åtgärda felet och köra vidare. Den sk turn-around-tiden är extremt kort.
- Programkoden går lätt att flytta mellan olika datortyper, endast den virtuella maskinen behöver bytas ut.
- När man håller på att lära sig språket kan man mycket enkelt stega sig genom ett program och direkt se vad varje sats åstadkommer.
- Man kan testa satser utan att skriva program.
- Kodstorleken blir mindre än i ett rent kompilerande system
- Prestanda blir bättre än i ett rent intepreterande system

Flyttar man miljön till en annan datortyp så behöver man bara byta ut den virtuella maskinen. All Smalltalkkod kompileras för den virtuella

[1] Med sparas avses här "accept" i något textfönster. Själva sparandet på sekundärminne sker tex via huvudmenyn.

maskinen som sedan, vid behov, i sin tur översätter till maskininstruktioner på den aktuella datorn.

Smalltalk har fått ett rykte om sig om att vara långsamt även om hastigheten har ökat sedan den dubbla kompileringen infördes. Trots detta används Smalltalk även i realtidssystem (som har stora krav på snabbhet). Snabbhet är ju inte endast en fråga om effektiv kod, det går att köpa snabbare processorer. Ett program måste inte vara den optimalt snabbaste lösningen, det räcker med att programmet är *tillräckligt snabbt* för uppgiften. Att få koden korrekt, lättarbetad och framför allt färdig i tid är ett mycket större problem. Genom att det är enkelt att skriva program så kan programmeraren koncentrera sig på att förstå problemet och hitta effektiva algoritmer. Därigenom får slutprodukten högre kvalitet.

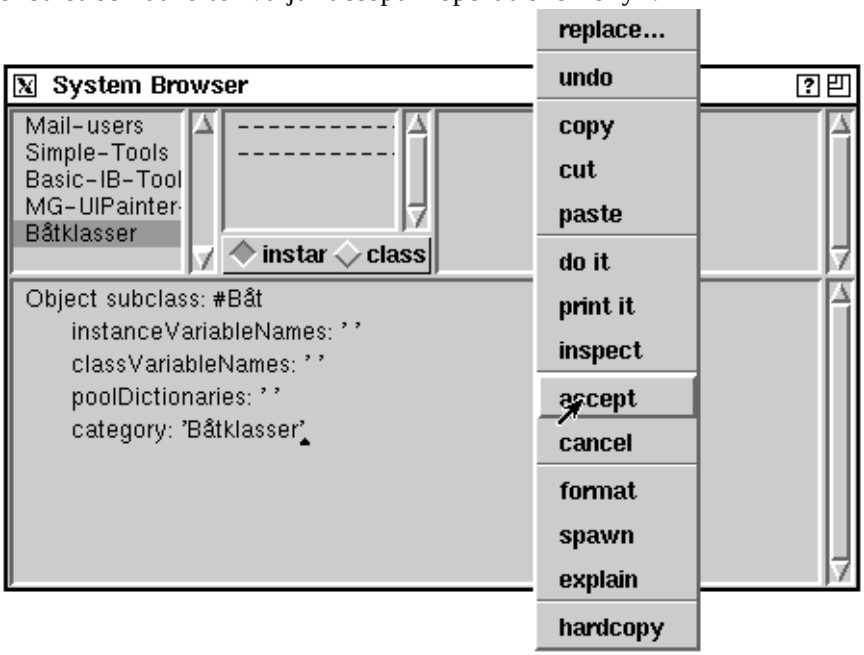
Dessutom har mycket stor möda lagts ned på att få den virtuella maskinen snabb, så ett stort Smalltalkprogram kan mycket väl bli snabbare än motsvarande C-program.

Deklarationer

Det finns Smalltalkklasser som hanterar deklarationer. Det enklaste sättet att deklarera klasser, attribut och metoder är att använda Browsern. Browsern ser i sin tur till att rätt metoder för deklarationerna blir anropade.

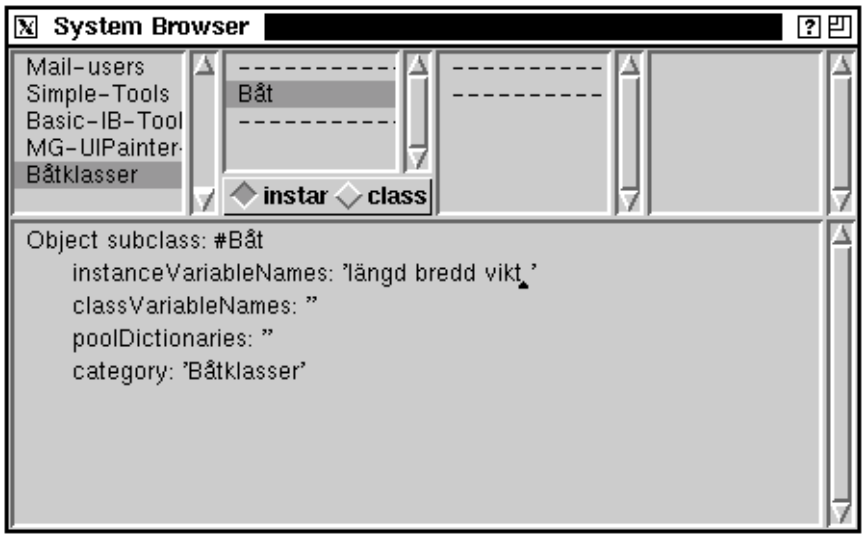
En ny klass deklarerar man genom att ange vilken klass den ska vara subklass till. Alla klasser i Smalltalk har en superklass. Högst upp i hierarkin finns klassen Object. Enklaste sättet att deklarera en ny

klass är att i browsern skriva in tex ”Object subklass #Båt” i metodfönstret och därefter välja ”accept” i operationsmenyn.



Figur 1.14 Båt deklareras som subklass till Object

Attribut deklarerar man genom att räkna upp deras namn i en sträng som är en av parametrarna till Object subclass:instanceVariableNames:class-VariableNanes:poolDictionaries:category:. Man skriver in de attribut man vill ha med i klassen och väljer därefter ”accept” i operationsmenyn.



Figur 1.15 Attributen längd, bredd och vikt deklareras

Variabelns typ bestäms först vid tilldelningen av ett värde till variabeln. Samma variabel kan alltså användas för heltal, flyttal, strängar och tom klasser!

Metoder deklarerar man genom att ange ett metodnamn i metodfönstret (det stora fönstret) i browsern och därefter välja **accept** i operationsmenyn. Varje gång man ändrar koden för metoden måste man ”spara” den med **accept**, glömmer man det så blir man tillfrågad om man vill spara ändringarna eller ej. Det går alltid att återgå till det senast sparade versionen genom att välja **cancel** i operationsmenyn. Detta gäller alla textfönster.

Operationerna (**accept**, **do-it** mfl) innebär att man gör anrop till Smalltalkklasser som utför det önskade. Dessa metoder skulle man lika gärna kunna anropa direkt för att lägga till attribut, skapa klasser mm, men det är oftast enklare att göra detta mha browsern.

Versaler och gemena är signifikanta i Smalltalk. Namnen n och N är alltså två olika namn.

Tilldelning

Tilldelning skrivs med := och fungerar som i Algol. Uttrycket till höger evalueras och tilldelas variablen till vänster. a := 17+4711 tilldelar a vär-

det 4728. Tilldelning av samma värde till flera variabler kan göras i en sats:

a := b:= c:= d:= e:= 17 tilldelar a, b, c, d och e värdet 17.

Val

Val skrivs som ett villkor som testas med ifTrue: och/eller ifFalse:.

Exempel: val

```
(a<b)
    ifTrue:[Transcript show: 'a är mindre än b']
    ifFalse:[Transcript show: 'b är mindre än eller lika med a']
```

Endera delen ifTrue:/ifFalse: kan utelämnas, endera kan stå först.

En detaljerad genomgång av vad som händer i Smalltalk när detta utförs kan vara intressant. Villkoret a<b returnerar en instans av en subklass till Boolean, nämligen True eller False. I både True och False finns metoderna ifTrue: och ifFalse: definierade. I True är metoden ifTrue: definierad så att blocket (det inom hakparenteserna) utförs, medan ifFalse: inte utför någonting alls. I False är metoderna definierade omvänt.

Block

Hakparenteserna [] i villkorsuttrycken markerar början och slutet på ett block och det som står ovan är alltså ett metodanrop ifTrue: med blocket som parameter. Block kan användas på ungefär samma sätt som lambda-uttryck i vissa Lispdialekter som Scheme, dvs de kan användas för att beskriva funktioner som kan skickas som parametrar till meddelanden mm.

Iteration

Iteration går att utföra på ett otal sätt i Smalltalk. Motsvarigheten till traditionella språks while-slinga ser ut så här:

```
[villkorsBlock]
    whileTrue: [satsBlock]
```

Så länge som villkorsBlock evalueras till true kommer satsBlock att utföras. Följande slinga skriver ut talen ett till tio:

```
| i |
i := 0.
[i<10]
    whileTrue: [i:=i+1.
        Transcript show: i printString].
```


Vill man vända på villkoret i villkorsBlock kan whileFalse**:** användas istället för whileTrue:.

Motsvarigheten till for-slingor går att utföra med någon variant av to:by:do:.

```
1 to: 10 by: 1 do: [:i | Transcript show: i printString]
```

gör samma sak som:

```
1 to: 10 do: [:i | Transcript show: i printString]
```

nämligen skriver ut talen ett till tio.

Metodanrop

När man vill använda en metod gör man ett *metodanrop* genom att skicka ett *meddelande* till ett objekt. Unära meddelanden är meddelanden utan parametrar. Exempelvis:

```
'Detta är en sträng' size
```

⇒*18*

String-objektet 'Detta är en sträng' tar emot meddelandet size och returnerar 18. Pilen ⇒, beskriver utmatning då Smalltalkuttrycket ovanför pilen utförs med **print it** i operationsmenyn. Vi använder också detta skrivsätt för att ange att utskrift i utmatningsfönstret (eng Transcript) sker.

Nyckelordsmeddelanden är meddelanden med parametrar:

```
'Detta är en sträng' at: 13
```

⇒*$s*

Strängen 'Detta är en sträng' tar emot meddelandet at: med parametern 13 och returnerar tecknet 's'. Strängen ses i det här fallet som en vektor och på position 13 finns tecknet 's'.

Binära meddelanden (som ser ut som de vanliga aritmetiska operatorerna), dvs objekt operator argument.

```
a := 1 * 2 + 3
```

Uttrycket kommer givetvis att ge a värdet fem, men

```
a := 1 + 2 * 3
```

ger a värdet nio! Orsaken är att binära meddelanden tolkas från vänster till höger, dvs 1 adderas till 2 som blir tre som i sin tur multipliceras med 3.

Prioritetsordning

Prioritetsordningen är:
1 Unära meddelanden
2 Binära meddelanden.
3 Nyckelordsmeddelanden.

Vid lika prioritet kommer satsen att evalueras från vänster till höger. Ett nyckelordsmeddelande slutar när satsen är slut, dvs när man kommer till en punkt (.), parentes ()), hakparentes (]), eller semikolon (;). Parenteser kan användas för att ändra på ordningen satserna utförs.

```
a := 1 + (2 * 3)
```
⇒*7*

Smalltalkexempel

Här följer ett exempel som visar hur en Smalltalkmetod ser ut. Metoden skriver ut en hailstonetalföljd i utskriftsfönstret. Talföljden fås så här: Tag ett tal, om talet är udda, multiplicera det med tre och lägg till ett. Om talet är jämnt dela det med två. Sluta när talet blir ett. Exempel: Talet 7 genererar serien 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1.

Namnet har talföljden fått av att talen växer och minskar i storlek likt ett hagelkorn (eng: hailstone) som i en storm virvlar upp och ned för att slutligen falla till marken. Talföljden är intressant för den terminerar för alla tal man har provat, men ingen har lyckats bevisa att den kommer att göra det för alla tal!

```
hailstone: anInteger
    | n |
    n := self.
    anInteger timesRepeat: [n = 1
        ifTrue:
            [Transcript show: 'Klart'; cr.
            ^n]
        ifFalse:
            [n odd
                ifTrue: [n := n * 3 + 1]
                ifFalse: [n := n // 2].
            Transcript show: n printString; cr]].
    Transcript show: anInteger printString , ' varv utan att nå ettan'; cr.
    ^n
```

Vill du provköra hailstoneexemplet kan du införa metoden i klassen Integer! I Smalltalk går det att göra tillägg och förändringar av själva systemet, men det ska naturligtvis göras med omsorg. Ändringar av systemet görs helt på egen risk, det kan i värsta fall resultera i att filen som allt sparas i (image) blir obrukbar. Tillägg kan göras, men ska i så

fall göras med största försiktighet. Hailstone-metoden är ett riskfritt tillägg. När du har skrivit in metoden och gjort ”accept” skriver du:

```
7 hailstone: 100
```

i arbetsfönstret, målar över allt med markören och väljer **do it** i operations-menyn. Då körs programmet och talföljden skrivs ut i utskriftsfönstret.

Här följer en detaljerad genomgång av metoden:

```
hailstone: anInteger
```

Metodnamnet är ”hailstone: ”. Kolon markerar att det finns en parameter (anInteger) till metoden.

```
| n |
```

Temporära variabler deklareras genom att innesluta dem mellan lodräta streck (|). Här deklareras den temporära variabeln n.

```
n := self.
```

n tilldelas värdet av mottagaren, heltalet sju i exemplet. Punkt skiljer satser åt.

```
anInteger timesRepeat: []
```

Blocket upprepas anInteger antal gånger.

```
(n=1)
```

Om n=1 returneras true och blocket efter ifTrue: utförs, om false returneras utförs blocket efter ifFalse:.

```
Transcript show: 'Klart'; cr.
^n
```

Skriv ut ”Klart” och ett radbyte i utskriftsfönstret, returnera n.

```
n odd
    ifTrue: [n := n * 3 + 1]
    ifFalse: [n := n // 2]
```

Om n är udda så tilldelas n värdet av n*3 + 1 annars tilldelas n värdet av n heltalsdividerat med två.

```
        Transcript show: n printString; cr]
```

Värdet av n skrivs ut. De två sista satserna kommer aldrig att utföras med vårt anrop.

```
Transcript show: anInteger printString,' varv utan att nå ettan'; cr.
```

Texten ”100 varv utan att nå ettan” och ett radbyte skrivs ut.

```
^n
```

Värdet av n returneras.

Sammanfattning

Termer

Klass beskrivning av ett objekt med data och operationer på dessa data.

Objekt en komponent i systemet med eget privat minne och en uppsättning operationer.

Instansvariabel en del av ett objekts privata minne.

Metod en beskrivning av hur ett visst objekt ska utföra en av sina operationer. Motsvarande procedur, funktion, subrutin i andra språk.

Attribut värden som hör till ett objekt.

Instans objekt skapat utifrån en klass. Har eget minnesutrymme.

Meddelande en uppmaning till ett objekt att utföra en metod.

Mottagare det objekt som uppmanas att utföra en metod.

Polymorfi olika metoder kan ha samma namn.

Dynamisk bindning vilken metod som används avgörs av klasstillhörigheten vid exekveringen, inte vid kompileringen.

Statisk bindning vilken metod som ska användas avgörs redan vid kompileringen.

Arv egenskaper i en klass kan användas i subklasser.

Subklass (härledd klass) klass som ärver egenskaper från en superklass.

Superklass (prefixklass) klass varifrån egenskaper ärvs.

Virtuella metoder metoder vars användning bestäms först vid exekveringen, inte kompileringen. Motsvarar dynamisk bindning.

Browser fönsterbaserad programkodshanterare.

Virtuell maskin (objektsmaskin) maskinberoende del av smalltalksystemet som hanterar exekvering av kod.

Utskriftsfönster (eng Transcript) fönster för utskrift av systemmeddelanden och för egna utskrifter.

Nyckelord en identifierare med ett avslutande kolon, tex `nyckel:`.

Unärt meddelande ett meddelande utan argument, tex `0.5 cos`.

Binärt meddelande ett meddelande med precis ett argument, tex `2 + 3`.

Nyckelordsmeddelande ett meddelande med ett eller flera argument åtskilda med nyckelord. Tex `x between: 3 and: 4`.

Skräpsamling systemet städar automatiskt bort objekt som ej längre refereras.

Programeringsstil

Delegera ansvar till objekt.

Förbered för ändringar.

Engagera slutanvändarna i programutvecklingen.

Smalltalk

Metoder alla metoder är virtuella, dvs kan omdefinieras i subklasser.

Uttryck eller sats en sekvens av operationer som beskriver ett objekt.

Tilldelning görs med :=. Tex x := 1995.

Satsparentes satser åtskils med en punkt (.), tex x := 2. y := x * 3.

Prioritetsordningen är:
Unära meddelanden
Binära meddelanden.
Nyckelordsmeddelanden.
Vid lika prioritet kommer satsen att evalueras från vänster till höger. Ett nyckelordsmeddelande slutar när satsen är slut.

Block en beskrivning av en fördröjd sekvens av operationer, skrivs inom ett par av hakparenteser. Exempel: [78 + 90]. Kan bla användas för att skicka kodbeskrivningar som parametrar.

Val skrivs som villkor följt av ifTrue: eller ifFalse: där argumentet är ett block som beskriver vad som skall göras om villkoret är uppfyllt. Tex x >= y ifTrue: ['x störst'].

Klassdeklaration en ny klass konstrueras utifrån en existerande klass. Samtidigt deklareras den nya klassens attribut.

Returvärde från metod mha ^ följt av vad som skall returneras. Tex

```
hallo
    ^'Hej'
```

Temporär variabel en variabel som konstrueras för att mellanlagra värden under en viss sekvens av operationer. Deklareras mellan ett par av vertikala linjer. Tex | temp1 temp2 |.

Iteration eller upprepning med whileTrue:, to:do: eller to:do:by:. Ett blockargument beskriver vad som ska göras i slingan.
Tex [x < y] whileTrue: [x := x + 1]. 1 to: x by: 1/2 do: [:i | z := z + i].

Polymorfi metoder i olika klasser kan ha samma namn.
Tex #(1 2 3) contains: [:x | x odd]
och (10@10 extent: 20@15) contains: (12@12 corner: 14@13).

Övningar

1.1 Vad är det för skillnad på strukturerad programmering och objektorienterad programmering?

1.2 När har man störst fördelar av objektorienterad programmering?

1.3 Vilka är Smalltalks fördelar och nackdelar?

1.4 Beräkna följande i ett arbetsfönster:

a) 1+2*3
b) (1+2)*3
c) 100 factorial
d) pi/2 cos
e) (pi/2) cos
f) 3 max: 4
g) 2 * 3 factorial

1.5 Skriv kod som givet talen a, b och c

a) ger det minsta värdet
b) returnerar en symbol som indikerar vilken av variablerna som är minst.

1.6 Skriv kod i ett arbetsfönster som tabellerar temperaturer mellan 37°C och 42°C i Fahrenheit enligt $F = 9/5 \cdot C + 32$

1.7 Skriv kod i ett arbetsfönster som skriver ut multiplikationstabellen på tio rader.

1.8 Skriv kod i ett arbetsfönster som beräknar värdet av e där

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots$$

1.9 Översätt följande matematiska uttryck till Smalltalk

a) 2+3
b) 2*3+4
c) 2+3*4
d) 10+5/2*3/4
e) Skriv om d) så att resultatet blir ett flyttal

1.10 Skriv smalltalkuttryck som beräknar summan av heltalen mellan a och b genom att använda

a) Någon for-slinge-liknande konstruktion

b) Metoden whileTrue:
c) Det slutna uttrycket (a+b)(b-a+1)/2
d) Något annat sätt.

talk

2 Objektkonstruktion

Detta kapitel besvarar bland annat följande frågor:

- Hur definieras klasser?
- Hur skapas instanser av klasser?
- Hur konstrueras metoder?
- Hur anropas metoder?
- Vad är inspektorer och mutatorer?
- Hur kommer man fram till vad som ska ingå i en klass?

Centralt i Smalltalk är att konstruera *klasser* med *attribut* och *metoder*. Eftersom allting i Smalltalk är objekt är det grundläggande hur *klasser* och *instanser* av dem konstrueras. Att skapa nya objekt från klassbeskrivningarna kallas för *instansiering* av klasserna.

2.1 Klassdefinition

En klass är en beskrivning av ett objekts egenskaper. Ur en mer praktisk synvinkel kan man säga att en klass är den programmeringskod som definierar objektets attribut och beteende. I Smalltalk definieras en ny klass alltid[1] genom att till en befintlig klass ange att en ny subklass ska konstrueras. Man gör detta genom att till den existerande klassen, dvs *superklassen*, ge den nya klassens namn samt ange dess eventuella attribut i form av instansvariabler, klassvariabler och poolvariabler. Vi kommer i detta avsnitt endast ta upp instansvariabler och återkommer till de andra två typerna i avsnitt 4.5. Genom att en klass alltid konstrueras utifrån en viss given superklass kommer en ny klass alltid att bli subklass till någon i systemet existerande klass.

En klass i Smalltalk kan definieras genom att utgå från en generell klassmall där lämpliga delar, dvs superklass, klassnamn och attribut, ersätts med dem som är aktuella. Denna klassmall ser ut som följer

[1] Här skiljer sig Smalltalk från vissa andra objektorienterade språk

```
NameOfSuperclass subclass: #NameOfClass
    instanceVariableNames: 'instVarName1 instVarName2'
    classVariableNames: 'ClassVarName1 ClassVarName2'
    poolDictionaries: ' '
    category: 'name of category'
```

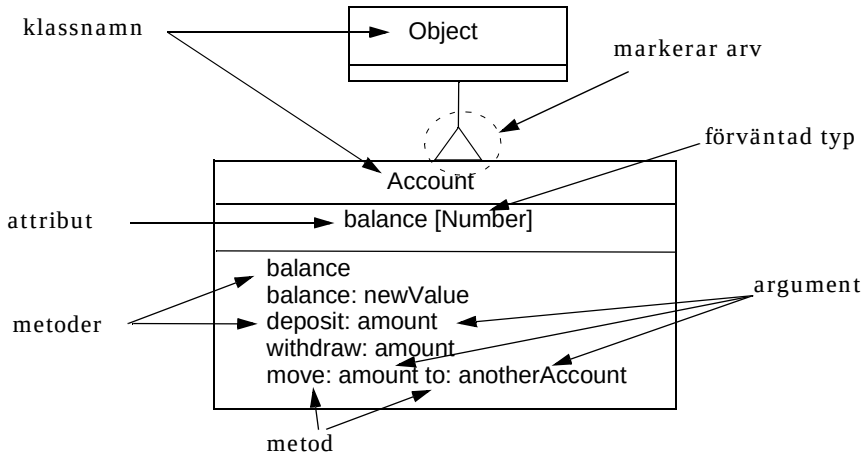
Som exempel konstruerar vi en klass för att hantera bankkonton. För att fokusera på praktiska tekniker i Smalltalk väljer vi att starkt förenkla modellen.

Enkel analys

Om vi utgår ifrån ett existerande bankkonto så kan vi definiera några nyckelord. Detta görs ofta vid analys med brainstorming, dvs först kastar man ur sig en massa begrepp som hör ihop, eller skulle kunna höra ihop med ett bankkonto och därefter försöker man besinna sig och se vad som ska vara med. En kort brainstorming-session skulle kunna ge följande: *konto, kassörska, bankomat, övertrassering, insättning, uttag, överföring, ränta, skuld, check, skatteflykt.*

När man därefter rensar upp bland idéerna brukar det ofta uppstå hetsiga diskussioner. Vad som ska ingå i en klass är ofta långt ifrån självklart och även med objektorientering kommer det att finnas flera möjliga lösningar.

I det här avsnittet kommer vi konstruera en klass Account med följande enkla klassbeskrivning.



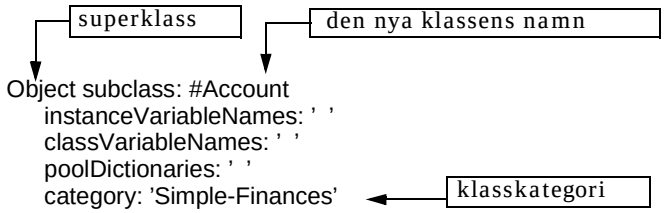
Figur 2.1 Klasshierarki: Account

I figuren ser vi att Account är subklass till Object och har en instansvariabel balance. Denna variabel ska vara av typen Number, dvs ett tal. Det finns fem metoder balance, balance:, deposit, withdraw: och move:to:

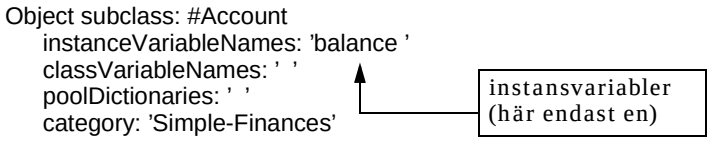
För att demonstrera hur en klass stegvis kan utvidgas och förses med attribut och operationer kommer vi att först definiera Account utan instansvariabler för att sedan i ett nästa steg lägga till instansvariabeln balance.

Praktik

Genom att i den ovan givna mallen ta bort alla attributnamn, ange Object som superklass, ge vår klass namnet Account, samt slutligen ge kategorinamnet Simple-Finances får vår klassdefinition följande utseende:



Men med denna definition har vi endast konstruerat en klass Account utan möjlighet att hantera specifika data för ett bankkonto. Vår avsikt är att beskriva en klass vars instanser ska representera balansen på bankkonton, dvs olika objekt ur klassen ska representera olika konton. Vi lägger därför till ett attribut balance, i form av en instansvariabel, till klassen. Vår nya definition blir som följer:



Därmed har vi lagt till den önskade instansvariabeln balance till klassbeskrivningen.

Om vi summerar vad som är sagt i detta avsnitt så kan vi beskriva det hela enligt följande:

Utgående från standardmallen för definition av klasser angav vi att en ny klass med namnet #Account skulle konstrueras som subklass till basklassen Object. Skrivsättet med tecknet # följt av klassnamnet är det sätt man anger att en ny klass med angivet namn ska konstrueras.

Vi angav också att klassen skulle ha en instansvariabel, balance, för att spara bankkontots saldo i.

Det finns också en ”väljare” med namnet category följt av strängen 'Simple-Finances'. Namnet anger klassens kategori. Detta ger oss möjligheter att gruppera klasser på andra sätt än genom arv.

2.2 Skapa objekt

Ett nytt objekt av en klass skapas, eller som det brukar kallas instansieras, genom att man ber önskad klass om en ny instans. Ofta sker detta genom att meddelandet new skickas till klassen och att resultatet av detta tilldelas en variabel, som nedan där variabel sätts att referera en ny instans av klassen KlassNamn.

```
variabel := KlassNamn new
```

För att skapa en instans av klassen Account gör vi:

```
myAccount := Account new
```

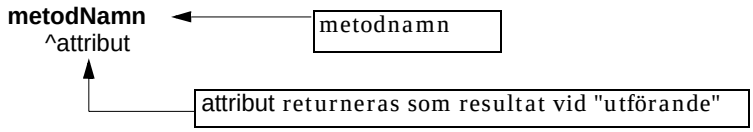
Efter detta refererar myAccount till en ny instans av klassen Account, men fortfarande utan möjlighet att komma åt eller ändra värdet av instansvariabeln balance, eftersom alla attribut i Smalltalk är skyddade från yttre åtkomst (se nedan).

2.3 Inspektorer eller Åtkomstmetoder

I Smalltalk är alla *attribut*, dvs *instans–*, *klass–* och *poolvariabler*, alltid åtkomliga innefrån en instans av den klass där de är definierade men skyddade från yttre åtkomst, dvs från läsning eller skrivning utifrån. Detta kallas för *inkapsling*. När man vill att dessa variabler ska vara åtkomliga även utanför det aktuella objektet måste *åtkomstmetoder (eng: access methods)* konstrueras.

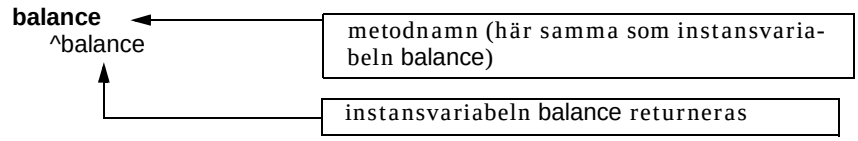
I språk som Simula och C++ kan man ange om egenskaper ska vara synliga utifrån och/eller i subklasser genom att uttryckligen ange egenskaperna som **`HIDDEN`** och **`PROTECTED`**, respektive **`public`** och **`private`**. Någon motsvarighet till detta finns inte i Smalltalk.

För att möjliggöra läsning av attributs värde måste *inspektorer* skrivas i klassen. Inspektorer skrivs enligt följande mall:

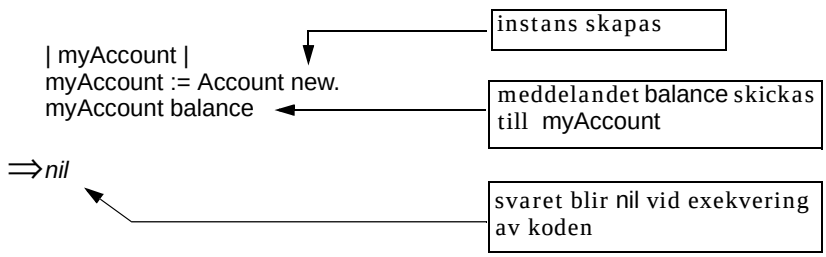


Där namnet på metoden (även kallad för selektorn) vanligen är det samma som attributets namn. Cirkumflextecknet (^), indikerar att det som följer ska returneras som resultat från metoden, i det här fallet attribut.

Nu skriver vi först en inspektor för den hittills enda instansvariabeln, vi väljer att ge metoden samma namn som instansvariabeln, nämligen balance.



Nu kan vi skapa en instans av klassen Account och testa metoden balance genom att utföra följande kod:



I det här kodavsnittet deklareras först den temporära variabeln myAccount. Sedan skapas en instans av Account som tilldelas till myAccount och slutligen skickas meddelandet balance, som i det här fallet ger nil i svar. Alla variabler får nämligen värdet nil från början.

Variabler

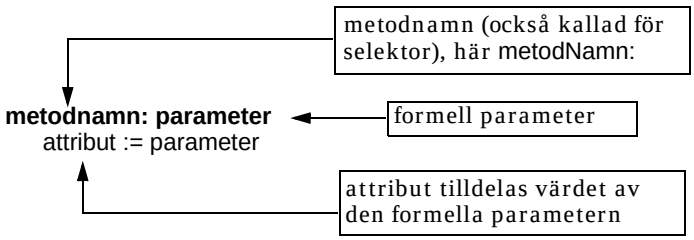
Deklaration av en metods temporära variabler sker genom att direkt efter metodhuvudet, mellan ett par av vertikala streck, räkna upp namnen på dessa variabler. En variabel har värdet nil innan något annat anges!

```
metodNamn
    | x y z temp |

    "...osv...".
```

I en temporärdeklaration är variablerna åtskilda av blanktecken eller annan typ av icke synligt skiljetecken, såsom tabulatortecken eller radslutstecken. Blanka, tabulatortecken och radbyten är inte signifikanta i Smalltalk. De temporära variablerna kan inte ges samma namn som de formella parametrarna och bör ej ges samma namn som instansvariabler.

2.4 Mutatorer eller uppdateringsmetoder

För att kunna förändra attribut måste klassen tillhandahålla metoder (mutatorer) för detta. Följande mall illusterar hur en typisk metod för att förändra värdet av ett attribut ser ut.



Observera att metodnamnet avslutas med ett kolon som sedan följs av den formella parametern, parameter, och att tilldelning av attributets värde sedan görs på vanligt sätt med tilldelningsoperatorn.

Med den ovan angivna mallen skriver vi nu en metod för att uppdatera värdet av instansvariabeln balance. Vi väljer precis som för åtkomstmetoden balance att ge den ett så förklarande namn som möjligt, nämligen balance:.

```
balance: newValue
    balance := newValue
```

Nu kan vi pröva de två instansmetoderna genom att i ett arbetsfönster, utföra följande:

```
| myAccount |
myAccount := Account new.
myAccount balance: 100.
myAccount balance
```

⇒*100*

Om vi utför denna kodsekvens kommer först myAccount att skapas, sättas till en ny instans av Account, värdet av instansvariabeln balance sättas till 100, via meddelandet balance: med myAccount som mottagare och därefter skickas meddelandet balance till myAccount, och resultatet returneras, i detta fall heltalsvärdet 100.

2.5 Metoddefinition

Nu ska vi titta lite närmare på hur metoder som inte är av ovan enkla åtkomsttyp kan konstrueras. Vi börjar med att definiera en ny metod som ska räkna upp saldot för aktuell instans med givet numeriskt argument.

```
deposit: amount
    balance := balance + amount
```

Om åtkomstmetoder finns så brukar man använda dessa. I deposit: ovan har vi inte utnyttjat att åtkomst- och uppdateringsmetoderna balance respektive balance: redan finns. För att göra detta så måste vi kunna skilja på om instansvariabeln eller metoden avses. Instansvariabeln refereras som ovan genom att enbart skriva namnet på densamma. För att referera till aktuellt objekt används pseudovariabeln self. Variabeln self kan också utnyttjas för att skicka meddelanden till det aktuella objektet, som i följande exempel:

```
deposit: amount
    | oldBalance newBalance |
    oldBalance := self balance.
    newBalance := oldBalance + amount.
    self balance: newBalance
```

Där vi först deklarerar två variabler oldBalance och newBalance, båda temporära. Sedan sätts oldBalance till resultatet av meddelandet balance skickat till den aktuella instansen. På näst sista raden ger vi newBalance värdet av att till oldBalance skicka meddelandet + med amount som argument. Slutligen skickar vi meddelandet balance: med newBalance som argument till aktuell instans.

Om vi utnytjar Smalltalks syntax och preferensregler så kan vi skriva om deposit: utan att använda temporära variabler:

```
deposit: amount
    self balance: self balance + amount
```

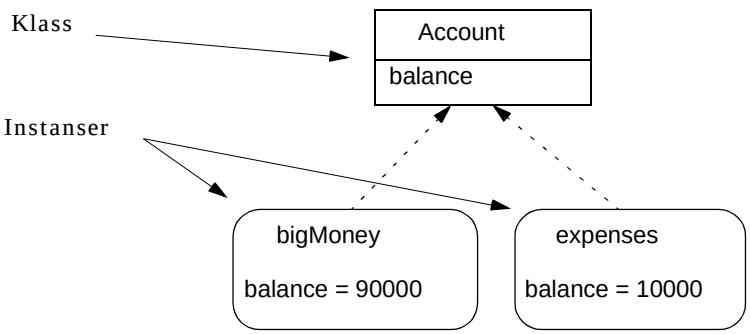
Dvs meddelandet balance: skickas till aktuell instans. Summan av resultatet från meddelandena balance och den formella parametern amount blir argument. Metoden withdraw: som tar ut pengar från konto skrivs i övning 2.1.

Vi prövar det hela genom att exekvera följande kod:

```
| bigMoney expenses transfer |
bigMoney := Account new.              bigMoney instansieras från Account
bigMoney balance: 100000.             bigMoney ges 100000 i startkapital
expenses := Account new.
expenses balance: 0.                  expenses ges 0 i startkapital

transfer := 10000.                    transfer används som 'slask-variabel'
bigMoney withdraw: transfer.          ta ut transfer från bigMoney
expenses deposit: transfer.    och deponera motsvarande på kontot expenses
expenses balance
```

⟹*10000*

Grafiskt kan situationen efter att koden utförts illustreras på följande sätt:



Figur 2.2 Instansdiagram: bigMoney och expenses är instanser av Account.

I figur 2.2 ser vi ett exempel på instansdiagram. De rundade rektanglarna representerar instanser av klassen Account.

Hittills har vi endast använt oss av metoder *utan* eller *med precis ett* argument, men för att i mer generella fall skriva strukturerade klasser så behöver vi kunna definiera metoder med flera argument. Vi ger i följande tabell helt kort några exempel på hur funktioner definierade matematiskt kan skrivas i Smalltalk. Observera att Smalltalks objektorienterade synsett ofta ger ett av de matematiska argumenten rollen av

ett objekt som är mottagare av det motsvarande Smalltalk-meddelande (som följdriktigt innehåller ett argument mindre än dess matematiska motsvarighet).

Matematiskt	Smalltalk	Kommentar
f(x)	f	x antar rollen av mottagare, dvs tillhör en klass som definierar metoden f, Anrop: x f
max(x, y)	max: y	Anrop: x max: y
min ≤ x ≤ max	between: min and: max	Metoden between:and: definierad i klassen som x tillhör. Anrop: x between: min and: max
integral(g, a, b)	integral: a to: b	g instans av klass av funktioner. Metoden ger integralen över angivet intervall. Anrop: g integral: a to: b

Figur 2.3 Matematiska- och Smalltalkuttryck

Metoder som tar argument som parametrar skrivs på formen (nyckelord1: parameter1 nyckelord2: parameter2 ...), dvs metoden består av ett eller flera nyckelord, vart och ett avslutat med ett kolon, följt av en formell parameter för vart och ett av dessa.

För att flytta pengar från ett konto till ett annat måste vi först ta dem från ett konto och sedan komma ihåg att sätta in dem på ett annat. Vi förbättrar Account genom att skriva en metod move:to: som flyttar en given summa pengar från det konto som är mottagare av meddelandet till ett annat som ges som argument.

```
move: amount to: anotherAccount
    "flyttar pengar från mig till kontot anotherAccount
    self withdraw: amount.
    anotherAccount deposit: amount
```

Den här metoden illustrerar bland annat att man kan skicka meddelanden till objekt givna som argument. Med hjälp av denna metod kan vi också skriva om exemplet där vi flyttade pengar från ett konto till ett annat, men den här gången med slaskvariabeln transfer ersatt med metoden move:to:

```
bigMoney expenses |
bigMoney := Account new.
bigMoney balance: 100000.
expenses := Account new.
expenses balance: 0.
bigMoney move: 10000 to: expenses.
expenses balance
```

⇒*10000*

2.6 Utvidgning av klass

Nu ska vi beskriva hur man kan utvidga en klass genom att lägga till nya instansvariabler och metoder.

Som Account är definierad så finns det inte i instanserna någon information om vilka konton de ansvarar för. Enda sättet att identifiera ett konto är genom namngivning av de respektive variablerna som pekar ut kontona (tex bigMoney). Detta är inte tillfredställande. Vi vill kunna identifiera konton genom att fråga kontot om dess nummer, detta bla för att ge smidigare stöd för transaktioner mellan konton och förbereda dem för lagring på sekundärminne. Vi utökar därför Account med attribut för identifiering av dess instanser.

Vi börjar enligt principen för stegvis förfining med att lägga till en instansvariabel number vars avsikt är att unikt identifiera det aktuella kontot. När vi har implementerat och testat detta så ger vi oss sedan på att konstruera mekanismer för transaktionhantering.

Klassen Account och dess instanser kommer efter de tillägg vi gjort fortfarande ha vissa konceptuella och tekniska brister. Exempelvis så måste användare av klassen både komma ihåg att göra initieringar samt ha kännedom om klassens interna struktur. Teknik för att åtgärda detta kommer vi beskriva i kapitel 5 och 11.

Först lägger vi till den nya instansvariabeln number genom att definiera om klassen på följande sätt:

```
Object subclass: #Account
    instanceVariableNames: 'balance number '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Simple-Finances'
```

För att ge möjlighet att läsa värdet av instansvariabeln number, konstruerar vi en inspektor:

```
number
    ^number
```

För att minska risken för att kontonumret oavsiktligt förändras och förtydliga andemeningen i koden så ger vi ingen metod för att ändra

enbart number. Istället konstruerar vi en speciell initieringsmetod som både sätter initialt saldo och kontonummer

```
initialBalance: aNumber accountNumber: anotherNumber
    "Initiera self med saldot aNumber och kontonummer anotherNumber"
    self balance: aNumber.
    number := anotherNumber        Det finns ingen metod för att uppdatera
                                   number, så vi får 'själva' göra tilldelningen.
```

Om vi skriver om initieringen av instanserna kan det se ut på följande sätt, där vi har valt att ge bigMoney och expenses kontonumren 998 respektive 999:

```
| bigMoney expenses |
bigMoney := Account new.
bigMoney initialBalance: 100000 accountNumber: 998.
expenses := Account new.
expenses initialBalance: 0 accountNumber: 999.
    ...osv...
```

En kort kommentar angående den givna lösningen. Vi kan ju fortfarande använda klassen på fel sätt genom att anropa initialBalance:accountNumber: mer än en gång, men i nuläget antar vi att endast snälla programmerare (eller de som är väldigt medvetna om vad de gör) använder klassen.

Transaktionslogg

Nu ska vi som avslutning på detta avsnitt implementera en enkel form av transaktionshantering. För att det inte ska bli för komplicerad så använder vi en enkel modell där varje av transaktion placeras i slutet av en kronologiskt ordnad lista, märkt med någon form av identifikation. Vidare kommer endast transaktioner som utförs med hjälp av metoden move:to: att loggas. Vi kommer använda en instans av OrderedCollection som bla ger oss möjligheter att lägga till nya element i slutet.

Först börjar vi med att lägga till den nya instansvariabeln genom att definiera om klasshuvudet.

```
Object subclass: #Account
    instanceVariableNames: 'balance number transactions '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Simple-Finances'
```

Sedan skriver vi en metod som när den utförs initierar instansvariabeln transactions till att bli en instans av OrderedCollection.

```
initializeTransactions
    transactions := OrderedCollection new
```

För att göra det möjligt att läsa transaktionsloggen så skriver vi också följande metod.

```
transactions
    ^transactions
```

Vi kommer att märka en viss transaktion genom att anropa följande rutin.

```
log: aNumber tagged: tag
    " lägger till en vektor med tag och aNumber sist i logglistan"
    self transactions add: (Array with: tag with: aNumber)
```

Här har vi använt meddelandet add: för att i slutet av logglistan lägga till en vektor med två komponenter, där komponenterna i tur och ordning är de formella parametrarna tag respektive aNumber. En instans av en vektor bestående av två element kan skapas genom att skicka meddelandet with:with: till klassen Array.

Nu skriver vi också om initieringsmetoden så att en loggning av det initiala saldot med märkning Initialt saldo sker på kontot. Vi har fortfarande ingen metod för att ändra kontonumret varför vi uttalat tilldelar number värdet av anotherNumber.

```
initialBalance: aNumber accountNumber: anotherNumber
    "initierar balansen med aNumber, kontotnumret med anotherNumber samt
    transaktionsloggen"
    self balance: aNumber.
    number := anotherNumber.
    self initializeTransactions.
    self log: aNumber tagged: 'Initialt saldo'
```

Slutligen lägger vi också till loggning av transaktioner vid flyttande av summor mellan konton. Vi har i det följande valt en enkel strategi där endast transaktioner som utförs mellan konton loggas. Ett annat alternativ vore att utföra loggning i metoderna withdraw: respektive deposit:.

```
move: amount to: anotherAccount
    "flyttar amount från self till anotherAccount"
    self withdraw: amount.
    self
        log: amount negated
        tagged: 'till: ', anotherAccount number printString.
    anotherAccount deposit: amount.
    anotherAccount
        log: amount
        tagged: 'från: ', self number printString
```

Vi har som synes valt att ta negativa värdet av amount (amount negated) i loggen för det konto vi tar pengarna ifrån, rubricerad med en sammanslagning av ordet 'till: ' (sammanslagning av strängar kan göras med ",") och det mottagande kontots nummer konverterat till en sträng (mha

printString). Det mottagande kontots logg uppdateras med rubricering 'från: ' istället.

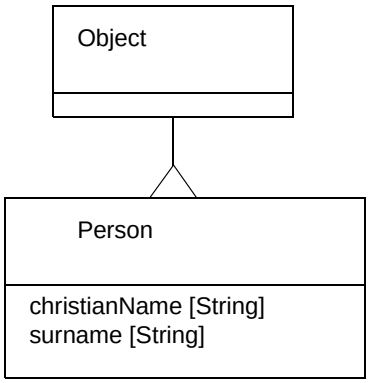
I figuren nedan visar vi steg för steg vad som händer med instansvariablerna balance och transactions efter det att satserna i den vänstra kolumnen utförs uppifrån och ner.

	bigMoney		expenses	
Uttryck	balance	transactions	balance	transactions
bigMoney := Account new.	nil	nil		
bigMoney initialBalance: 100000 accountNumber: 998.	100000	(#('Initialt saldo' 100000))		
expenses := Account new.			nil	nil
expenses initialBalance: 0 accountNumber: 999.			0	(#('Initialt saldo' 0))
bigMoney move: 10000 to: expenses.	90000	(#(Initialt saldo' 100000) #('till: 999' -10000))	10000	(#(Initialt saldo' 0) #('från: 998' 10000))
expenses move: 250 to: bigMoney.	90250	(#('Initialt saldo' 100000) #('till: 999' -10000) #('från: 999' 250))	9750	(#('Initialt saldo' 0) #('från: 998' 10000) #('till: 998' -250))

Figur 2.4 Penningtransaktioner m h a Account

2.7 Person exempel

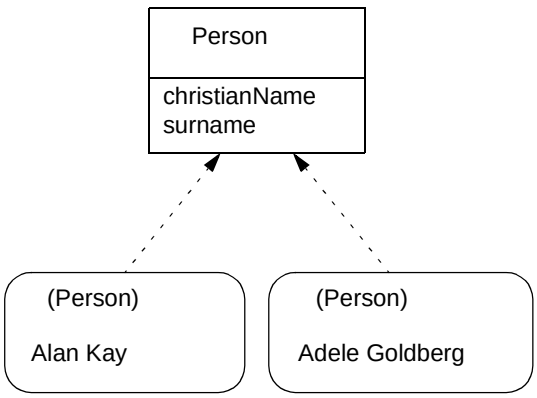
Vi ska som en repetition av kapitlet, konstruera klasser vars instanser representerar personer. Vi börjar med en enkel modell och antar att en person kan representeras med sitt för- och efternamn. I detta avsnitt kommer vi förutom att definiera klassen endast skriva åtkomstmetoder för instansvariablerna och två enkla metoder av annat slag.



Figur 2.5 Klasshierarki: Person

Figuren ovan visar den enkla modell som vi avser att konstruera, där vi väljer Object som superklass till vår klass Person. Klassen Person ska vidare definieras med två instansvariabler, en för förnamn och en för efternamn, både av typen String.

Nedan visar vi en situation där vi har konstruerat två olika instanser av klassen Person.



Figur 2.6 Instansiering: Person

Först klassdefinitionen:

```
Object subclass: #Person
    instanceVariableNames: 'christianName surname'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Simple-Finances'
```

Nu definierar vi metoder för att läsa respektive ändra värdet av instansvariabeln christianName. Först definierar vi en metod som ger möjlighet att utifrån läsa variabelns värde

```
christianName
    ^christianName
```

och en metod som möjliggör förändring av densamma.

```
christianName: aString
    christianName := aString
```

Vi lämnar metoderna för att läsa respektive uppdatera instansvariabeln surname till övning 2.2.

Då Smalltalk är otypat indikerar ofta namnet på den formella parametern (här aString) dess förväntade klasstillhörighet. Som vi tidigare sett är det inte nödvändigt att ge åtkomstmetoden samma namn som den aktuella instansvariabeln, så en åtkomstmetod som returnerar aktuellt efternamn skulle kunna skrivas som följer:

```
familyName
    ^surname
```

En annan möjlighet vore att tillhandahålla metoder med olika namn för att bland annat tillmötesgå olika klienters krav, se kapitel 11.

Nu visar vi hur den här klassen kan användas för att konstruera instanser av Person. Alla utmatningar här sker i utskriftsfönstret.

Konstruktion av instans

En instans skapas genom att skicka meddelandet new till önskad klass.

```
| scientist |
scientist := Person new.
scientist christianName: 'Albert'.
scientist surname: 'Einstein'.

Transcript show: 'Physicist: ', scientist christianName, ' ',
                 scientist surname.
Transcript cr
```

⇒Physicist: Albert Einstein

Inget hindrar oss från att ändra delar av ett objekt

Vi kan ansätta ett värde för ett visst attribut och senare ändra detta som i följande exempel, där vi ändrar efternamnet från Einstein till Pascal.

```
| person |
person := Person new.
person christianName: 'Albert'.
person surname: 'Einstein'.

person surname: 'Pascal'.

Transcript show: 'Name: ', person christianName, ' ', person surname.
Transcript cr
```

⇒Name: Albert Pascal

eller att "glömma" initieringen av vital del av objektet

I följande exempel ”glömmer” vi att ange något efternamn .

```
| person |
person := Person new.
person christianName: 'Albert'.

Transcript show: 'Name: ', person christianName, ' ', person surname.
Transcript cr
```

Det senare resulterar i ett felavbrott. Orsaken till detta är att vi försöker konkatenera person surname, som ju är nil, med en sträng. Detaljer om fel och felhantering kommer i kapitel 12. Olika sätt att hindra variabler från att förbli oinitierade beskrivs i avsnitt 4.4.

Sammanfattning

Termer

Instansiering att skapa nya objekt från en klass.

Instansvariabel variabel med ett eget värde för varje instans av en klass.

Inspektor en metod som returnerar ett attribut.

Mutator en metod som direkt ändrar ett attribut och inget annat.

Inkapsling att dölja ett objekts inre struktur från dess (externa) gränssnitt.

Metod beskriver vad ett objekt ska göra om motsvarande meddelande skickas till det.

Metodnamn en metods namn, exklusive parametrar. Tex abs, transactions, +, min:, beteween:and: eller move:to:.

Selektor synonymt med metodnamn. Ibland avses metodnamnets symboliska form, tex #between:and:.

Klass en beskrivning av objekt med data och operationer på dessa objekt.

Superklass den klass varifrån variabler och egenskaper ärvs.

Subklass en klass som ärver variabler och egenskaper från en existerande klass.

OMT Object Modeling Technique en analys- och designmetod vars syntax används i figurerna i boken.

Metodik

Brainstorming för att hitta möjliga egenskaper hos en klass.

Iterativ programutveckling.

Programeringsstil

Använd alltid *åtkomstmetoder,* även för attribut i den egna klassen.

Skriv kommenterer direkt efter huvudet i alla metoder utom åtkomstmetoder som endast returnerar eller ändrar attributet med samma namn.

Smalltalk

Nya *klasser* defineras genom att ange en ny subklass till en existerande klass.

Nya *attribut* läggs till genom att räkna upp dem i klassdefinitionen.

Alla *attribut* är skyddade för åtkomst utifrån.

Variabler *initieras* till nil.

metoder med parametar skrivs: nyckelord1: parameter1 nyckelord2: parameter2 ...

self används för att referera till det aktuella objektet.

Klassdefinition en ny klass konstrueras utifrån en existerande klass. Samtidigt deklareras den nya klassens attribut.

```
Superklass subclass: #NyKlass
    instanceVariableNames: 'instansVariabel1 instansVariabel2'
    classVariableNames: 'KlassVariabel1 KlassVariabel2'
    poolDictionaries: ''
    category: 'Klasskategori'
```

Instansiering en instans skapas genom att skicka ett meddelande till en klass.

```
instans := Klassnamn new
```

Variabel är temporär, global, instans-, klass- eller poolvariabel. Får värdet nil från början.

Formell parameter representerar ett argument i en metod. Kan ej tilldelas nytt värde.

Övningar

2.1 Skriv metoden Account>>withdraw: som gör det omvända mot Account>>deposit:, dvs minskar saldot med givet argument istället.

2.2 Skriv motsvarigheter till metoderna christianName och christianName: för att läsa respektive uppdatera instansvariabeln surname i klassen Person med följande namn:

a) surname som returnerar en pekare till instansvariabeln
b) surname: som uppdaterar värdet av densamma

2.3 Body Mass Index, BMI, används för att avgöra om personer är normalviktiga. BMI = vikt i kg /(längd i meter)^2. Normalviktiga har BMI mellan 20 och 25. Deklarera en klass som innehåller en instansvariabel vikt och för längd samt en metod som returnerar BMI.

2.4 Tjebysjevpolynom kan beräknas ur en rekursionsformel:
$T_n = 2xT_{n-1} - T_{n-2}$, $T_0 = 1$, $T_1 = x$.
Skriv en metod i klassen Integer som tar x som parameter och returnerar värdet av T_n.

2.5 Definiera en klass Student med följande instansvariabler förnamn, efternamn och utbildningslinje. Skriv lämpliga inspektorer och mutatorer.

2.6 Skriv en klass Artikel med instansvariablerna artikelnummer, artikelnamn, antalILager, antalSålda och pris.

a) Skriv inspektorer och mutatorer
b) Skriv metoderna lagerVärde och försäljningsIntäkter som ger lagrets totala värde respektive hur mycket som är sålt av den aktuella varan vid en given tidpunkt

2.7 Följande deluppgifter går ut på att skriva en klass Punkt.

a) Beskriv klassen med lämplig representation och lämpliga åtkomstmetoder.
b) Skriv metoden dist: enPunkt som beräknar avståndet till en given annan punkt.
c) Skriv metoden translatedBy: enPunkt som beräknar den nya Punkt som man får om man translaterar med enPunkt uppfattad som en vektor.
d) Skriv metoden scaledBy: amount som returnerar en ny Punkt som man får om man skalar om argumentet (uppfattat som en vektor) med talet amount.

2.8 Följande deluppgifter bygger vidare på Punkt-klassen för att definiera en klass Rektangel.

a) Skriv klassens grundläggande delar genom att ange dess representation och åtkomstmetoder.
b) Skriv Punkt-metoden extent: enPunkt som returnerar en Rektangel.
c) Skriv metoden area.
d) Skriv metoden center.
e) Skriv metoden moveBy: enPunkt som translaterar en Rektangel utan att skapa en ny.
f) Skriv metoden copy som gör en kopia av en Rektangel.

3 Smalltalk fundament

Detta kapitel besvarar bland annat följande frågor:

- Hur jämför man objekt i Smalltalk?
- Hur tar man reda på ett objekts klass?
- Hur tar man reda på vilka meddelanden man kan skicka till ett objekt?
- Vad händer egentligen vid tilldelning?
- Hur skrivs tal, tecken, strängar, symboler och vektorer?
- Vilka typer av meddelanden kan skickas?
- I vilken ordning utförs meddelanden?

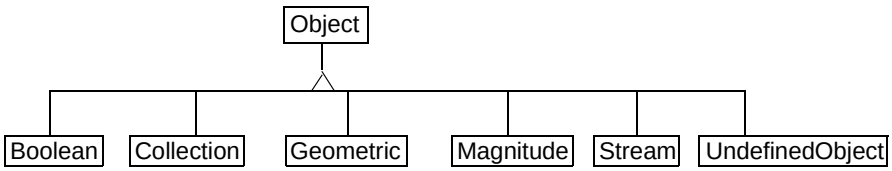
I det här kapitlet beskriver vi den grund som *språket* Smalltalk är byggt på. Vi inleder med att beskriva de viktigaste delarna av klassen Object som är *rotklass*, dvs superklass till alla andra klasser i systemet. Detta innebär att alla objekt i systemet förstår meddelanden som har metoder definierade i Object.

I och med att man som programmerare får tillgång till all källkod i smalltalksystemet så kan man omdefiniera systemklasser genom att ändra befintliga metoder, även om detta ofta inte är att rekommendera, skriva nya metoder och definiera nya subklasser till redan existerande klasser.

En liten kärna av primitiver, anrop till den virtuella maskinen (objektmaskin), används för de mest basala operationerna som addition och multiplikation av flyttal. Detta görs bland annat av prestandaskäl, men även därför att någonstans måste själva uträkningen bli gjord. Även metoderna för dessa operationer är fullt möjliga att omdefiniera. Kort sagt Smalltalk är inte som andra språk!

3.1 Object och UndefinedObject

Alla klasser är subklasser till klassen Object. Därmed ärver också alla klasser metoder därifrån. Detta medför att alla objekt förstår meddelanden vars metoder hör hemma i Object. Däremot är det inte säkert att alla objekt använder Object´s metod då de tar emot motsvarande meddelande! Detta beror på att en klass alltid kan definiera en egen metod med samma namn som superklassens metod. En orsak till att omdefiniera är att metoder definierade i Object ibland saknar mening för vissa objekt, men detta är mer undantag än regel.



Figur 3.1 Klasshierarki: Object och några viktiga subklasser

Det väsentliga är att alla objekt förstår meddelanden definerade i Object och att omdefinierade metoder har likvärdig betydelse som Objects metod.

UndefinedObject

En beskrivning av klassen Object kräver en beskrivning av klassen UndefinedObject.

UndefinedObject instansieras aldrig av användarna utan alla använder en av systemet från början given instans: nil. Denna ”variabel” representerar som i många andra språk ett icke-värde, den tomma pekaren eller helt enkelt avsaknaden av något värde. Det vore också orimligt att tilllåta att nya instanser av denna klass skulle kunna skapas då värdet nil används för att kontrollera om en variabel inte har antagit något specifikt värde. Om detta var tillåtet skulle två nil-objekt kunna vara olika. Försöker man i alla fall skapa en instans av UndefinedObject resulterar det i ett felavbrott.

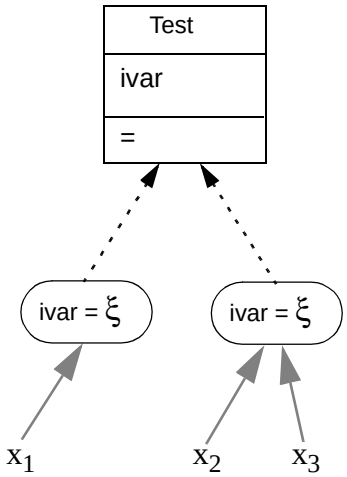
Objektidentitet och jämförelser

Att jämföra olika objekt med varandra och agera beroende av resultatet är fundamentalt i alla program. Viktigt är då att definiera vad som avses med att två objekt är identiska eller när de är lika i en svagare mening.

Samma (==) eller lika (=)

I Smalltalk finns två metoder för jämförelse, en för att undersöka om två objekt är identiskt lika, referenslikhet (==), och en för att undersöka om objekten har samma värde, värdelikhet (=).

Antag att vi har följande situation där cirklarna representerar två instanser av en viss klass Test, och x_n är variabler som refererar till



någon av dessa. Klassen har en instansvariabel ivar som för båda instanserna antar värdet ξ. Om vi i figuren antar att klassen har definierat =-likhet som att de respektive instansernas instansvariabler är lika så kommer bl a följande gälla i figuren $x_1 = x_1$, $x_1 = x_2$ och $x_2 = x_3$, då alla antar samma värde. Däremot är inte x_1 **==** x_2 men x_2 **==** x_3, då de första två pekar ut olika instanser men de senare pekar ut samma instans. Skönsvärdet för =, eller det som metoden Object>>= definierar, är ekvivalent med metoden ==.

Om vi exempelvis konstruerar två instanser från ekvivalenta vektorer enligt följande:

```
| a b |
a := #(1 2 3).
b := #(1 2 3).
```

så kommer en jämförelse med = ge oss svaret att a och b är lika, dvs:

```
    a = b
⇒true
```

Men då a och b ej är samma objekt så kommer en jämförelse med == ge false som resultat, dvs:

```
    a == b
⇒false
```

Metoden == är så pass central att den inte ska omdefinieras. En viktig konsekvens av att = kan omdefinieras är att = inte säkert är kommutativt, dvs det kan gälla att x1 = x2 ger true men att x2 = x1 ger false som svar! I extremfallet kan det tom vara så att x1 = x2 ger ett svar men att x2 = x1 resulterar i ett felavbrott! Allt hänger förstås på hur programmeraren har implementerat metoden. För det mesta så gäller att = är ekvivalent med ==. Det är ju praktiskt att 10 = 10 och 10 == 10.

Olika värden (~=) eller olika identitet (~~)

Motsatsen till = och == är ~= respektive ~~, dvs ~= svarar på om värdet av objekten är olika och ~~ om objekten ej är identiska. Exempel:

```
    | a b c |
    a := 17.
    b := 17.
    c := 4711.
    a ~= b.
⇒false
    a ~=c.
⇒true
    a ~~ b.
⇒true
```

Nil (isNil) eller ej (notNil)

Det finns två viktiga metoder för kontroll av ett objekts identitet nämligen isNil och notNil. Som namnen antyder så svarar det första meddelandet på frågan om ett objekt är nil och det andra på om det inte är nil. Den första av dessa två metoder frågar ett objekt om det är lika med nil, dvs x isNil är ekvivalent med både x = nil och x == nil.

Givet en variabel x kan vi undersöka om den är nil på följande sätt:

a) x isNil
b) x = nil
c) x == nil

Att meddelandet går att skriva om med båda formerna av likhet beror på att nil är ett unikt definierat konstant objekt. Analogt kan vi skriva om notNil mha av test på likhet eller olikhet. Även om man kan skriva dessa tester på olika sätt så är oftast meddelandena isNil eller notNil att föredra, eftersom meddelanderna är unära blir det enklare att infoga i komplicerade uttryck samt inbjuder till en mer objektorienterad stil. I systemet förekommer de flesta formerna flitigt.

3.2 Klasstillhörighet

Smalltalk är ett dynamiskt typat språk, där ingen variabel tilldelas någon typ vid deklarationen utan först vid instansieringen. I många sammanhang kan det vara värdefullt att veta vilken klass, eller gren av arvsträdet, som ett objekt tillhör. Det finns många olika sätt att göra detta på.

Meddelande	Kommentar	Exempel
class	Objektets klass	17 class ⟹*Integer*
species	I stort sett ekvivalent med class. Det finns några undantag där species returnerar något annat än metoden class. Ett exempel är om en viss konkret klass är maskinberoende, då kan species returnera en abstrakt maskinoberoende klass.	17 species ⟹*Integer*
isMemberOf:	Är det mottagande objektets klass det samma som argumentets.	17 isMemberOf: Integer ⟹*true*
isKindOf:	Är det mottagande objektets klass identisk med argumentet eller någon av dess subklasser.	17 isKindOf: Integer ⟹true
superclass	Det mottagande klassobjektets superklass.	Integer superclass ⟹Number
isString	Är objektet en sträng?	'sjutton' isString ⟹true
isInteger	Är det ett heltal?	17 isInteger ⟹true

Figur 3.2 Meddelanden för att avgöra klasstillhörighet

Man bör vara sparsam med att använda meddelandet isKindOf: då det oftast inte ger en objektorienterad lösning och dessutom kan ta lång tid att utföra. Ibland kan det också vara meningsfullt att söka ett visst objekts typ mer frikopplat från dess arvsträd. I figur 3.3 finns exempel på metoder för detta.

Meddelande	Kommentar	Exempel
respondsToArithmetic	Förstår mottagaren meddelanden av aritmetisk typ, t ex +, -, * och /? Detta meddelande ger true som svar för instanser av ArithmeticValue med subklasser.	
isSequenceable	Är mottagaren instans av en klass av mängdtyp där mängden definierar en ordning mellan elementen? Observera att ordningen är definierad av listan och inte av relationer mellan elementen!	| x y | x := #(1 2 3). x isSequenceable. ⇒*true* y := x asSet. y isSequenceable ⇒false
respondsTo:	Förstår mottagaren det meddelande som ges som argument till metoden?	47 respondsTo: #abs. ⇒*true*

Figur 3.3 Meddelanden som avgör mottaglighet

3.3 Tilldelning

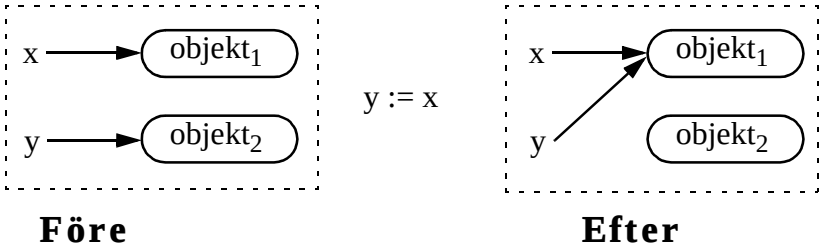
En variabel tilldelas värde med tilldelningsoperatorn (:=). Innan en variabel har tilldelats ett värde så har den värdet nil.

```
| x | "deklaration av den temporära variabeln x"
"Nu har x värdet nil"
x := 10 "x tilldelas värdet 10"
```

Man kan i huvudsak tänka sig två olika betydelser av tilldelning: *kopieringssemantik* vilket innebär att vid tilldelning kopieras värdet av den högra sidan och därefter kommer de två värdena vara oberoende av varandra eller *referenssemantik* vilket innebär att variabeln på vänster sida inte bara får samma värde som utrycket på höger sida utan också referar till detta objekt, detta innebär också att en förändring i någon av dem påverkar den andra. Smalltalk använder referenssemantik vid tilldelning.

Semantik vid tilldelning

Följande figur illustrerar detta:



Figur 3.4 Referenstilldelning

I figur 3.4 ser vi att om y via tilldelning sätts till x så kommer därefter x och y peka på samma objekt ($objekt_1$). Det som y tidigare pekade ut, $objekt_2$, kommer tas om hand av skräpsamlaren.

Detta innebär också att om vi utför satserna uppifrån och ner i kolumnen uttryck i följande tabell, så får variablerna a och b de värden som skrivs i respektive a- och b-kolumn:

uttryck	a	b
| a b |	nil	nil
a := {1 2}	{1 2}	nil
b := a	{1 2}	{1 2}
a add: 3	{1 2 3}	{1 2 3}
b add: 4	{1 2 3 4}	{1 2 3 4}

Figur 3.5 Förändring via meddelanden

Observera att tilldelning ändrar objektpekaren, medan meddelandesändnig bevarar objektets identitet. Figur 3.6 illustrerar detta.

uttryck	x	y
x := 10	10	nil
y := x	10	10
x := x + x	20	10

Figur 3.6 Förändring via tilldelning

3.4 Kopiering

Ofta behöver man skapa en kopia av ett befintligt objekt. Vi ska nu titta på Smalltalks olika sätt att göra detta. Vi beskriver vad de olika sätten får för konsekvenser, bland annat vid jämförelser mellan original och kopior eller kopiorna sins emellan. De olika formerna av kopiering är också, som vi ska se, nära knutet till de sätt som vi kan jämföra samt identifiera objekt.

Kopia

- copy

  ger en ny instans som är likadan som mottagaren.

  Skönsvärdet är shallowCopy följd av postCopy (se nedan), eftersom metoden copy är definierad på följande sätt i objekt:

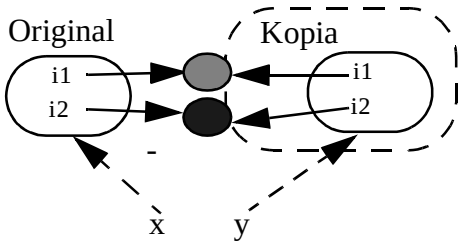
```
copy
    ^self shallowCopy postCopy
```

  Däremot gör inte postCopy i Object någonting.

Kopia som delar instansvariabler

- shallowCopy

  ger en kopia av mottagaren där eventuella instansvariabler delas mellan original och kopia



Figur 3.7 Effekten av y := x shallowCopy

Egen anpassning

- postCopy

  definierar vad som måste göras för att åstadkomma copy efter det att shallowCopy har använts.

  Typiskt betyder det att en klass som har instansvariabler definierar postCopy med superklassens postCopy följt av att alla instansvariabler

kopieras, vilka annars kommer delas med den instans från vilken kopian gjordes.

Det är viktigt att vara medveten om att man i en egen klass kan behöva omdefiniera postCopy för att ge en djupare kopiering än vad metoden copy gör från början!

3.5 Litterala konstanter

För att göra det enklare att använda några grundläggande typer av objekt använder Smalltalk sex olika så kallade *litterala uttryck*. Då dess värden alltid är unika och specifika objekt kallas de också för litterala konstanter. Några av dessa konstanter kan konstrueras genom att man på vanligt sätt via en klass konstruerar en ny instans. Men några av dem kan endast konstrueras mha dess litterala form. Exempelvis vore det svårt att konstruera tal eller teckenkonstanter på annat sätt.

Tal

Tal representeras på naturligt sätt, dvs ett antal siffror eventuellt föregångna av ett minustecken. Decimaltecken skrivs med en punkt. Ett flyttal måste ha minst en siffra till vänster och en till höger om decimalpunkten. Som exempel är följande tal möjliga att skriva i Smalltalk:

```
17, -4.678, 602299993035875123462144
```

Tal kan skrivas på annat sätt än decimalt (som förstås om vi inte uttrycker något annat). En talbas anges med hjälp av prefixet 'r' (för radix). Radix är i intervallet 2 till 256.

Följande är exempel på tal i olika radix:

Matematiskt	Smalltalk
1000_2	2r1000
7654_8	8r7654
$A3FF_{16}$	16rA3FF
-23.5_8	8r-23.5

Figur 3.8 Tal i olika baser

Ett tal skrivs normalt ut i sin decimala form t ex vid **print it** i ett arbetsfönster. För att skriva ut ett tal med en annan bas används meddelandet printStringRadix: där önskad radix anges som argument.

Uttryck	Resultat	Kommentar
2r1000 printStringRadix: 2	'1000'	2r1000 i basen 2
8r1000 printStringRadix: 8	'1000'	8r1000 i basen 8
16rFFFF printStringRadix: 64	'Fvv'	16rFFFF i basen 64

Figur 3.9 Utmatning med olika bas

Tal kan också skrivas med exponent genom att inkludera suffixet 'e' (exponent) eller 'd' (dubbel precision) följt av önskad exponent. Man kan också använda 'd' som suffix för att indikera dubbel precision, se figur 3.10.

Smalltalk	Resultat	Kommentar
2e3	2000	$2 \cdot 10^3$
2e-3	(1/500)	$2 \cdot 10^{-3}$
2.0e3	2000.0	$2.0 \cdot 10^3$
2.0e-3	0.002	$2.0 \cdot 10^{-3}$
123456789.0e-10	0.0123457	Observera trunkeringen
123456789.0d-10	0.0123456789d	Med Double får vi bättre precision
0.123456789	0.123457	Trunkering
0.000000001d	1.0d-9	Dubbel precision
0.123456789d	0.123456789d	Tillräcklig precision

Figur 3.10 Exponentiering och större precision

Tecken

En teckenkonstant skrivs med ett dollar-tecken ('$') som prefix.

```
$a  "tecknet a"
$%"tecknet %"
$*  "tecknet *"
$$  "tecknet $"
```

Strängar

En sträng skrivs inom ett par av omslutande apostrofer('), som i:

```
'Det här är en sträng'
'Apostrofer i strängar skrivs med två apostrofer. Som här: Smalltalk''s way!'
```

Symboler

Symboler är etiketter som består av ett unikt namn-objekt, t ex ett klassnamn. (För varje symbol i systemet finns endast en instans, dvs om en symbol används på olika ställen så är objektidentiteten ändå densamma).

En symbolisk litteral inleds med ett nummertecken (#), och man kan använda apostrofer för att t ex hantera blanktecken och andra tecken som annars ej är möjliga att använda.

```
#Symbolname
#AnotherSymbol
#'A symbol with spaces'
```

Litteral vektor

En instans av Array kan också konstrueras med hjälp av en litteral beskrivning. En sådan vektor skrivs med hjälp av ett nummertecken (#) följt av innehållet inneslutet inom ett par av runda parenteser.

En litteral vektor kan också nästlas med andra litteraler, dvs det är möjligt att direkt definiera litterala vektorer som består av andra litterala objekt.

```
#(1 2 3 77 98 12)
#('a string' #symbol 454)
#(45 #(1 'test' 34 #('inner' -667) 77 'end'))
```

Byte-vektorer

Den sista typen av litterala former är instanser av ByteArray som skrivs med hjälp av inledande symboltecken följt av numeriska byte-värden, dvs heltal mellan 0 och 255, inneslutna inom ett matchande par av hakparenteser.

```
#[123 255 0 10 67]
```

3.6 Meddelanden

I Smalltalk sker informationsutbyte mellan objekt via meddelanden. Ett meddelande är en uppmaning till ett objekt att utföra en av sina metoder. En meddelandesändning består av tre delar, en mottagare, en metod-selektor och noll eller flera argument. Alla meddelanden resulterar i att ett objekt returneras som resultat till meddelandesändaren.

Typer

Det finns tre olika typer av meddelanden i Smalltalk nämligen:

unära meddelanden

Ett unärt meddelande består av en mottagare och en selektor, men inga argument.

I figur 3.11 följer några typiska exempel på unära meddelanden

Uttryck	Resultat	Kommentar
-1995 abs	1995	absolutbeloppet
0.67 sin	0.620986	sinus
Date today	19 June 1995	dagens datum
Float pi.	3.14159	π som flyttal
Double pi	3.1415926535898d	π i dubbel precision
7 negated.	-7	negativa värdet
'test' size	4	storleken av mottagaren
#(1 2 3) last	3	sista elementet

Figur 3.11 Unära meddelanden

binära meddelanden

Ett binärt meddelande har också en mottagare och en selektor, men kräver precis ett argument.

I figur 3.12 nedan illusteras några enkla binära meddelanden.

Uttryck	Resultat
2 * 5	10
3 < 5	true
'xyz' ~~ #'xyz'	true
'Konkat', 'enering'	'Konkatenering'
89 \\ 4	1
true & false	false

Figur 3.12 Binära meddelanden

nyckelordsmeddelanden

Nyckelordsmeddelanden är den tredje och sista typen av meddelanden. Det har (som alla meddelanden) en mottagare, ett eller flera nyckelord samt ett argument per nyckelord.

I figur 3.13 ges några typiska exempel på nyckelordsmeddelanden.

Uttryck	**Resultat**
Transcript show: 'Nyckel'	"Ordet Nyckel skrivs i utskriftsfönstret"
#(nil) at: 1 put: Browser	#(Browser)
Delay forSeconds: 2	"En instans av Delay skapas"
'byte' changeFrom: 1 to: 0 with: 'ut'	'utbyte'
'1.# x*z' match: '1.3 x identiskt med z'	true

Figur 3.13 Nyckelordsmeddelanden

Preferens- eller prioriteringsregler

I ett Smalltalk–uttryck utförs unära meddelanden först, därefter binära meddelanden och sist nyckelordsmeddelanden. Från modellen med mottagare och meddelanden följer att ett uttryck i huvudsak utförs från vänster till höger – då ju ett meddelande alltid skickas till en mottagare. Denna ordning kan ändras med hjälp av parenteser.

I figur 3.14 ger vi ytterligare några exempel på meddelanden av de tre

Uttryck	Mottagare	Selektor	Argument	Typ[a]	Resultat
17 odd	17	odd	inga	u	true
-88 negated	-88	negated	inga	u	88
2 reciprocal	2	reciprocal	inga	u	(1/2)
3 + 4	3	+	4	b	7
12 / 9	12	/	9	b	(4/3)
3 max: 4	3	max:	4	n	4
2 between: 1.3 and: 3.6	2	between:and:	1.3 och 3.6	n	true
Array with: 1 with: 2 with: 3	Array	with:with:with:	1, 2 och 3	n	#(1 2 3)

Figur 3.14 Exempel på Smalltalks tre meddelandetyper

a. Typ av meddelande där u = unärt, b = binärt och n = nyckelords

olika slagen. Här har vi endast givit exempel på uttryck bestående av en av de tre olika typerna i taget.

Vi sammanfattar Smalltalks prioriteringsordning i figur 3.15:

1 meddelanden utförs från *vänster till höger*
2 med en *strikt prioritetsordning* mellan de olika meddelandena enbart baserad på dess typer, där
   - *unära* meddelanden utförs först
   - *binära* sedan
   - *nyckelordsmeddelanden* utförs sist
3 prioritetsordningen kan ändras med hjälp av *parenteser*

Figur 3.15 Prioritetsordning

Observera att prioritetsordningen innebär att Smalltalk inte har någon prioritet mellan meddelanden inom en meddelandekategori, exempelvis så har * samma prioritet som + i aritmetiska uttryck! Så

```
2 + 3 * 4
⇔ 5 * 4
⇔ 20
```

men

```
2 + (3 * 4)
⇔ 2 + 12
⇔ 14
```

3.7 Syntax

Med det tidigare resonemanget om meddelanden blir en beskrivning av Smalltalks syntax väldigt enkel. Det enda som tillkommer är en dikussion om meddelanden i sekvens och (det syntaktiska sockret) kaskadmeddelanden.

Meddelanden i sekvens

Varje meddelande till ett objekt ger ett svar, ofta bara i form av en referens till mottagaren, men ibland till ett helt nytt objekt. Därmed kan vi omedelbart skicka ett nytt meddelande till resultatet.

Dvs

```
o2 := o1 m1.
o3 := o2 m2.
o4 := o3 m3.
```

är ekvivalent med

```
o4 := o1 m1 m2 m3.
```

som är ekvivalent med

```
o4 := ((o1 m1) m2) m3.
```

Exempel: Trigonometri

Beräkna $\sqrt{\sin^2 20 + \cos^2 70}$ där argumenten till de trigonometriska funktionerna är angivna i grader

Ledning:

I Smalltalk anges vinklar i radianer. Använd degreesToRadians för att konvertera grader till radianer. Meddelandena för de trigonometriska funktionerna i uttrycket ovan är de unära meddelandena sin respektive cos och kvadrering görs med det unära meddelandet squared. Slutligen roten ur fås med det unära meddelandet sqrt.

Lösning:

Vi använder meddelanden i sekvens och utnyttjar att unära meddelanden har högre prioritet än binära, samt använder parenteser där det behövs

```
(20 degreesToRadians sin squared +
    70 degreesToRadians cos squared) sqrt
```

⟹*0.483689*

Kaskadmeddelanden

En vanligt situation är att man till ett och samma objekt vill skicka flera olika meddelanden i följd. Vi kan antingen göra detta genom att tilldela objektet till en variabel och sedan i olika satser skicka meddelandena till den eller genom att utnyttja s k kaskadmeddelanden.

Vilket medför att

```
o m1.
o m2.
o m3.
```

också kan skrivas

```
o m1; m2; m3.
```

Exempel: Kaskadmeddelande

Givet följande temporära variabler med tilldelningar

```
head := 'Kaskadmeddelanden' asUppercase.
bread := 'kan man klara sig utan
men kan vara bekvämt!'.
```

Skriv ut variablerna i utskriftsfönstert genom att först göra en radframmatning, skriva head, ytterligare en radframmatning, sedan bread och slutligen ytterligare en radframmatning. Först utan att använda kaskadmeddelanden och därefter med kaskadmeddelanden

Lösning:

a) meddelanden i sekvens

```
Transcript cr.
Transcript show: head.
Transcript cr.
Transcript show: bread.
Transcript cr
```

b) kaskadmeddelanden

```
Transcript cr; show: head; cr; show: bread; cr
```

⇒

Observera att

```
    o m* mi; mj
⇔   | x |
    (x:= o m*) mi.
    x mj
```

där **m*** är noll eller flera godtyckliga meddelanden, $\mathbf{m}_i$ samt $\mathbf{m}_j$ är godtyckliga meddelanden.

3.8 Styrstrukturer

Till skillnad mot många andra språk så finns det i Smalltalk inga speciella syntaktiska konstruktioner för att hantera styrstrukturer och därmed besläktade ting. I Smalltalk hanteras också dessa på vanligt sätt med hjälp av objekt och meddelandesändning.

I det här avsnittet ger vi en kort framställan av Smalltalks viktigaste objekt för att hantera styrstrukturer, nämligen *block* och *boolean*. För en fördjupad beskrivning hänvisas till kapitel 6.

Block

Ett *block-uttryck* representerar en fördröjd sekvens av operationer. Ett sådant uttryck skrivs genom att man innesluter vanliga Smalltalksatser inom ett par av hakparanteser, som i:

```
[4 + 5]
```

För att satserna i ett block ska utföras så måste vi skicka meddelandet value till det. Blocket returnerar resultatet av den sista satsen.

```
[4 + 5] value
```
⟹*9*

Vi kan också referera till variabler i omgivningen.

```
| x |
x := 1.
[x := x + 5] value.
```
⟹*x = 6*

Eller binda ett block till en variabel.

```
| x b |
x := 1.
b := [x := x + 5].
b value.
```

Den inramade delen kan skrivas om på följande sätt:

```
b := [ x + 5].
x := b value.
```

I båda fallen får vi resultatet:

⟹*x = 6*

Ett block kan också användas flera gånger.

```
| x b y |
x := 1.
b := [x + 5].
x := b value.
y := b value.
```
⟹*x = 6 och y = 11*

Block med argument

Det är möjligt att ge ett block upp till 255 argument. Argument till ett block måste deklareras genom att efter den vänstra hakparantesen deklarera de formella parametrarna med namn föregånget av ett kolon. Deklarationen av argument och satserna ska åtskiljas med ett vertikalt streck (|).

Till ett block med ett argument används meddelandet value: *parameter* där argumentet *parameter* kommer att bindas till den formella parametern.

Observera att det inte går att tilldela en formell parameter ett nytt värde!

Nu kan vi återigen skriva om föregående exempel med hjälp av ett block med *ett* argument:

```
| x b y |
b := [:arg | arg + 5].
x := b value: 1.
y := b value: x.
```

⇒*x = 6 och y = 11*

För att utföra ett block med två argument används meddelandet value:value:.

Så om vi generaliserar lösningen i det trigonometriska exemplet på sidan 74 mha ett block får vi följande beskrivning:

```
b := [:x :y | (x degreesToRadians sin squared +
            y degreesToRadians cos squared) sqrt].
b value: 20 value: 70
```

⇒*0.483689*

Ett block kan innehålla flera Smalltalkuttryck. Resultatet av den sista satsen returneras.

```
| traceAndConcatenate aString |
traceAndConcatenate := [:stringOne :stringTwo |
        Transcript show: stringOne; cr.
        Transcript show: stringTwo; cr.
        stringOne, stringTwo].
aString := traceAndConcatenate value: 'xxx' value: 'yyy'.
traceAndConcatenate value: aString value: aString
```

Som ger utskriften

⇒

xxx
yyy
xxxyyy
xxxyyy

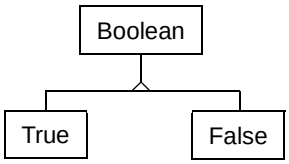
Den sista satsen i exemplet returnerar strängen 'xxxyyyxxxyyy'

För en utförligare beskrivning av block som sådana se kapitel 6.

Logiska uttryck och villkorssatser

Viktigt i alla programmeringspråk är möjligheten att uttrycka villkor. I Smalltalk finns (självklart) också en sådan möjlighet. En väsentlig skillnad mellan Smalltalks och andra språks logiska uttryck är att i Smalltalk är även de logiska sanningsvärdena (true och false) objekt av

första ordningen, dvs de är också instanser av klasser med klassbeskrivningar.



De två variablerna true och false tillhör klasserna True respektive False, båda subklasser till Boolean. Dessa logiska icke förändringsbara variabler intar en viss särställning då inga andra instanser av dess klasser bör konstrueras. Om man gör det så bryter systemet snabbt ihop då logiska jämförelser resulterar i olika sanningsvärden.

Ett logiskt utryck genereras som resultat av vissa meddelandesändningar. Detta har vi sett flera exempel på tidigare i texten, bl a där vi tidigare i kapitlet jämförde olika objekt eller frågade om dess ”typ” var av något visst slag.

Villkorssatser

En villkorssats har en av följande fyra former:

1 boolean ifTrue: aBlock
där boolean kan vara ett godtyckligt logiskt uttryck och aBlock är ett block utan argument.

```
| a b |
a := 5.
b := 4.
a > b ifTrue: ['a störst']
```

⇒*'a störst'*
Om villkoret är falskt returneras nil.

```
| a b |
a := 3.
b := 4.
a > b ifTrue: ['a störst']
```

⇒*nil*

2 boolean ifFalse: aBlock
analogt med ifTrue: men blocket utförs bara om villkoret är falskt

3 boolean ifTrue: aBlock ifFalse: alternativeBlock
om boolean är sann utförs aBlock annars alternativeBlock.

```
"max"
| a b |
a := 17.
b := 15.
```



```
a >= b
        ifTrue: [a]
        ifFalse: [b]
```

⇒*17*

Som framgår utförs alltid något av de villkorade blocken och resultatet retureneras från villkorsatsen.

4 boolean ifFalse: aBlock ifTrue: alternativeBlock
analog med ifTrue:ifFalse:

Logiska uttryck

Det finns också en en del andra viktiga meddelanden för att hantera styrstrukturer och logiska operationer, som t ex

- logisktUttryck not

  logiskt icke

  true not

⇒*false*

  Icke kan vi också uttrycka mha av meddelandena ~= eller ~~.

- logisktUttryck & annatLogisktUttryck

  För *konjunktion* eller *logiskt och* kan vi använda meddelandet &

  3 < 4 & (5 < 9)

⇒*true*

- logisktUttryck | annatLogisktUttryck

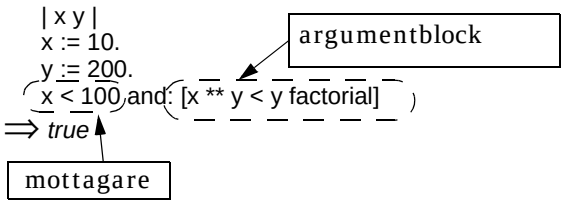
  För *disjunktion* eller *logiskt eller* kan vi använda meddelandet |

  'test' size < 2 | (3 factorial between: 2 and: 10)

⇒*true*

- logisktUttryck and: ettBlockSomReturnerarLogisktVärde

  Samma som & men med lat evaluering, dvs argumentet, givet av ett block, utförs endast om det första villkoret är sant.



  Argumentblocket utförs endast om mottagaren är true.

- logisktUttryck or: ettBlockSomReturnerarLogisktVärde

  Samma som | men med lat evaluering

Villkorliga slingor

Som ni har sett kombineras logiska uttryck med block för att hantera styrstrukturer. Villkorliga slingor använder främst block men använder logiska uttryck för att kontrollera dem.

Följande meddelanden är grundläggande vid hantering av villkorade slingor:

1 aBlock whileTrue: evaluationBlock

så länge som true returneras från aBlock utför evaluationBlock

```
| i |
i := 1.
[i < 10]
    whileTrue: [Transcript show: i printString, ' '.
                i := i + 1]
```

⟹ *1 2 3 4 5 6 7 8 9*

2 aBlock whileFalse: evaluationBlock

så länge som false returneras från aBlock utför evaluationBlock

```
| i |
i := 1.
[i >= 10]
    whileFalse: [Transcript show: i printString, ' '.
                i := i + 1]
```

⟹*1 2 3 4 5 6 7 8 9*

3 aBlock whileTrue

fortsätt så länge som true returneras från aBlock

```
| i |
i := 1.
[Transcript show: i printString, ' '.
i := i + 1.
i < 10] whileTrue
```

⟹*1 2 3 4 5 6 7 8 9*

4 aBlock whileFalse

fortsätt så länge som false returneras från aBlock

```
| i |
i := 1.
[Transcript show: i printString, ' '.
i := i + 1.
i >=10] whileFalse
```

⟹*1 2 3 4 5 6 7 8 9*

Slingor

Det finns en uppsjö av olika metoder för itereration över mängder, intervall och liknande typer av objekt. Vi illustrerar några typiska sådana i detta avsnitt och återkommer till andra senare.

Upprepning

Vi beskriver några typiska metoder för att upprepa ett block ett givet antal gånger.

- antalGånger timesRepeat: block

  För att upprepa satserna i ett block ett givet antal gånger kan meddelandet timesRepeat: användas.

```
| a b times sum current |
a := 7.
b := 15.
times := b - a + 1.
sum := 0.
current := a.
times timesRepeat: [sum := sum + current.
                    current := current + 1].
sum
```

⇒*99*

- startVärde to: stoppVärde do: ettArgumentsBlock

  Smalltalks motsvarighet till en vanlig for-slinga heter to:do: där mottagaren och det första värdet är ett numeriskt värde. Det sista argumentet är ett block med en formell paramater. Parametern till blocket representerar aktuellt värde i slingan. Steget är ett.

```
| sum a b |
sum := 0.
a := 7.
b := 15.
a to: b do: [:i | sum := sum + i].
sum
```

⇒*99*

- startVärde to: stoppVärde by: steg do: ettArgumentsBlock

  Om man vill ange ett steg (kanske negativt, flyttal eller på bråkform) så får man istället använda meddelandet to:by:do: där argumentet till nyckelordet by: är steget.

```
15 to: 7 by: 2 negated do: [:i | sum := sum + i].
sum
```

⇒*55*

  Om vi vill stega baklänges med steget ett så kan vi alternativt skriva:

```
(a to: b) reverseDo: [:i | sum := sum + i]
```

Men då måste parenteserna finnas med!

- intervall inject: värde into: tvåArgumentsBlock

  Ett av de mer kraftfulla konstruktionerna vid itererande är meddelandet inject:into:. Här kan mottagaren vara ett intervall, det första argumentet ett startvärde och blocket beskriver hur värdet ska beräknas. Det första argumentet i blocket beskriver det ackumulerade värdet och det andra det aktuella värdet i slingan.

```
(7 to: 15) inject: 0 into: [:total :part | total + part]
```

⟹*99*

Med dess hjälp kan vi skriva fakultetsfunktionen på följande sätt:

```
(1 to: 5) inject: 1 into: [:x :y | x * y]
```

⟹*120*

Behållare

Nu beskriver vi också några typiska konstruktioner för att iterera över ett objekt av behållar- eller vektorliknande typ.

Alla metoder som följer kan användas med någon form av behållare som mottagare men vi har använt vektorer i exemplen.

- behållare do: ettBlock

  Iterera över mottagaren och utför argumentblocket med alla element.

```
| sum |
sum := 0.
#(7 8 9 10 11 12 13 14 15) do: [:anElement | sum := sum + anElement].
sum
```

⟹*99*

- behållare collect: ettBlock

  Konstruera en ny behållare, iterera över mottagaren och stoppa in resultaten av att utföra argumentblocket på alla element.

```
#(7 8 9 10 11 12 13 14 15) collect: [:anElement | anElement squared]
```

⟹*#(49 64 81 100 121 144 169 196 225)*

- behållare select: ettBlock

  Skapa en ny behållare bestående av de objekt i mottagaren som uppfyller villkoret som beskrivs i blocket.

```
#(7 8 9 10 11 12 13 14 15) select: [:anElement | anElement odd]
```

⟹*#(7 9 11 13 15)*

- behållare detect: ettBlock

  Sök efter första element i mottagaren som uppfyller blockets villkor.

  ```
  #(7 8 9 10 11 12 13 14 15) detect: [:anElement | anElement > 10
                                                  and: [anElement even]]
  ```

⇒*12*

- behållare reject: ettBlock

  Skapa en ny behållare men rata de objekt i mottagaren som uppfyller villkoret som beskrivs i blocket.

  ```
  #(7 8 9 10 11 12 13 14 15) reject: [:anElement | anElement > 10
                                                  and: [anElement even]]
  ```

⇒*#(7 8 9 10 11 13 15)*

- behållare inject: startVärde into: ettBlock

  Blocket som har två formella parametrar beskriver hur nästa värde ska beräknas utgående från det startvärde som ges efter nyckelordet inject:. Den första parameten till blocket "håller i" det ackumulerade värdet och den andra representerar aktuellt värde i behållaren.

  ```
  #(7 8 9 10 11 12 13 14 15) inject: 0 into: [:total :part | total + part squared]
  ```

⇒*1149*

  De ingående objekten behöver självklart inte vara numeriska:

  ```
  #('konk'  'at' 'enera') inject: '' into: [:total :part | total , part ]
  ```

⇒ *'konkatenera'*

3.9 Variabler

I Smalltalk finns det variabler knutna till klasser, temporära variabler, globala variabler samt några speciella variabler, ibland kallade pseudovariabler.

Temporära variabler

En temporär variabel (lokal variabel) är en variabel som deklareras lokalt i en viss sekvens av kod, t ex en metod.

Deklaration

Temporära variabler deklareras inom ett par av vertikala streck. Vid deklaration av flera samtidigt ska de åtskiljas med mellanslag, tabbar eller returtecken.

```
| temp1 enTill enAnnanTemp |
```

Livslängd

En temporär variabel lever tills dess koden i den aktuella lexikala omgivningen är utförd, oftast tills dess metoden är slut.

Global variabel

En global variabel är en variabel som är åtkomlig och förändingsbar överallt i systemet. Alla globala variabler befinner sig i Smalltalks systemkatalog som har namnet Smalltalk (som själv är global och därigenom också ingår i sig själv!).

Några exempel på globala variabler är Transcript, nyss nämnda Smalltalk, Processor och alla klasser i systemet.

Deklaration

En global variabel kan kostrueras genom att lägga den i systemkatalogen.

```
Smalltalk at: #MinGlobalaVariabel put: nil                    initialt värde nil
```

En klass skapas normalt genom att systemets verktyg används och en ny global variabel skapas ofta interaktivt genom att man låter systemet fråga om vad en viss variabel är, vid t ex exekvering i ett arbetsfönster.

Livslängd

Om man inte speciellt ber om det tas inte en global variabel bort. En klass tar man normalt bort med hjälp av en browser och menyalternativet **remove**. Andra globala variabler tas bort på följande sätt:

```
Smalltalk removeKey: #MinGlobalaVariabel
```

Klassbundna variabler

Som vi har sett finns det en hel del variabler som är bundna till klasserna och deras instanser. Det finns också variabler nära knutna till objekten som används då metoder skrivs.

Instans och klassvariabler

Instansvariabler har vi sett prov på tidigare i texten och en djupare beskrivning av dessa och klassvariablerna görs i avsnitt 4.3.

self och super

För att referera till den instans som utför ett viss metod använder vi oftast den speciella- eller pseudovariabeln self. För att referera en metod definierad högre upp i klasshierarkin används pseudovariabeln super.

thisContext

Variabeln thisContext kan användas för att referera till aktuell exekveringsomgivning. Den här variabeln behöver man sällan eller aldrig använda om man inte skriver inspektionsverktyg, avlusare eller liknande.

Speciella konstanta variabler

De finns också tre speciella konstanta värden: nil, true och false. Variabeln nil representerar en ickepekare eller ickevärde i Smalltalk. Alla variabler i systemet får värdet nil från början, De andra två konstanta värdena, true och false, har uppenbar mening.

3.10 Klassen Association

Klassen Association beskriver ett nyckel-värde par. Den används av systemet för att hålla ihop två element där det ena betraktas som nyckel och det andra som värde. För att skapa en association använder man vanligen det binära meddelandet ->, som alla objekt förstår. Mottagaren av meddelandet blir nyckel och argumentet värde.

En association kan vara lämplig att använda om man t ex vill returnera två objekt från en metod. Det ena får då vara nyckel och det andra värde i associationen.

För att komma åt attributen key och value använder man meddelandena med samma namn. Det går också att ändra attributen med meddelandena key: och value:

Som exempel på en association skapar vi en vars nyckel är symbolen #time och värde är aktuell tid. Därefter skriver vi ut den i utskriftsfönstret.

```
| ass |
ass := #time -> Time now.
Transcript show: 'key=', ass key printString, ' value=' , ass value printString
```

⇒*key=#time value=11:51:20 am*

3.11 Ett objekts presentation på skärmen

Viktigt i de flesta applikationer är de sätt som program, data eller objekt presenterar sig på skärmen. I den här boken beskriver vi inte hur man konstruerar grafiska interaktiva gränssnitt och presentationer av objekt, detta återkommer vi till i boken Interaktionsprogrammering i Smalltalk. Här visar vi hur man i Smalltalk med relativt enkla medel kan förändra en klass textuella representation på skärmen. För att göra det så kan man antingen definera om metoden printString, eller lite mer generellt printOn:, som i sin tur används av bl a printString. Metoden printString används bl a vid menyalternativet **print it**.

Omdefinition av presentationsmetoderna

printString

Metoden printString ska returnera en sträng som beskriver hur det aktuella objektet ska presentera sig textuellt på skärmen. En klass som representerar cyklar skulle kunna ha följande metod printString:

```
printString
    ^'jag är en cykel'
```

printOn:

Lite mer generellt kan man istället definiera om printOn:. Detta kräver dock kunskaper om strömmar, vilka vi återkommer till i kapitel 10.

displayString

För en lite mer avancerad typografisk beskrivning av ett objekt skickas meddelandet displayString till det. Från dess metoder kan man returnera en presentation som istället för strängar använder texter, dvs med t ex fetstil eller färg. Denna metod används av bland annat listvyer, så metoden ska helst inte returnera en för omfattande beskrivning. Om man inte definierar om metoden i en subklass så används Object's metod som helt enkelt returnerar resultatet av printString.

3.12 Namnkonventioner

Om du tittar närmare på de kodexempel som finns i boken så märker du att vi konsekvent har använt inledande versal för alla klassnamn respektive gemen bokstav för allt annat. Detta är inte någon slump utan följer de konventioner som finns i Smalltalk.

Konventioner

I Smalltalk finns det inte så många givna syntaktiska eller namngivningsregler. Däremot finns det en hel del konventioner (som nästan är lagar) för hur namn väljs. En viktig anledning till att det finns konventioner för detta är att om dessa följs så blir det mycket enklare att läsa och förstå (speciellt andras) kod och program. Figur 3.16 sammanfattar kort dessa konventioner:

Typ av namn	Inleds med versal/ Obligatoriskt	Exempel
Klasskategori	Ja/Nej	Magnitude-Numbers
Klass	Ja/Ja	Float
Klassvariabel	Ja/Ja	Pi
Global variabel	Ja/Ja	Transcript
Pool-variabel	Nej/Nej	cr
Instansvariabel	Nej/Nej	numerator
Temporär variabel	Nej/Nej	tmp
Metod-kategori	Nej/Nej	factorization and divisibility
Metod	Nej/Nej	factorial

Figur 3.16 Namnkonventioner

3.13 Exempel: person med flera förnamn

Vi fortsätter med personexemplet från avsnitt 2.7 sidan 52. Vi gör två små förändringar: christianNames i pluralform och surname utbytt mot familyName. Den första förändringen beror på att vi senare vill återspegla att en person vanligen har flera olika förnamn.

Klassdefinition: Name

```
Object subclass: #Name
    instanceVariableNames: 'christianNames familyName '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'bok-1-person'
```

Trots att vi har valt att ändra christianName till christianNames och surname till familyName så kan klassen fortfarande ”vara lojal mot” eller anpassad för Person genom att fortfarande tillhandahålla metoder christianName och surname

Åtkomstmetoder

```
christianNames
    ^christianNames

familyName: aString
    familyName := aString
```

Vi lämnar de likartade metoderna christianNames: och familyName till övning 3.5.

```
surname
    ^self familyName

surname: aString
    ^self familyName: aString
```

I inledningen nämnde vi att Name skulle kunna hantera personer som hade flera namn. För att enkelt åstadkomma detta utnyttjar vi en instans av OrderedCollection med en lista av en viss persons förnamn. För att förenkla beskrivningen så antar vi att tilltalsnamnet alltid ligger först i listan. För användare av den tidigare versionen av Name väljer vi att också behålla de ”gamla” metoderna christianName och christianName:, och vi ser till att de utåt sett beter sig som förut.

Med dessa antaganden blir det enkelt att behålla det gamla kategorinamnet för att läsa namn. Vi använder helt enkelt meddelandet first, som returnerar första komponenten i en sekvensierbar lista.

```
christianName
    ^self christianNames first
```

Att skriva metoden för christianName: blir lika enkelt. Vi använder här istället meddelandet addFirst: som till en given instans av OrderedCollection placerar objektet som ges som argument först, med redan befintliga

element ”skiftade” ett steg bakåt. Om listan är tom så kommer argumentet bli listans enda element.

```
christianName: aString
    ^self christianNames addFirst: aString
```

Hur kan denna klass förenkla hanteringen av namn?

Först så måste vi ändra definitionen av Person till att istället använda en instansvariabel för att både hantera för- och efternamn och vidare måste vi se till att initiera alla instansvariabler i Name på lämpligt sätt!

Vi gör som tidigare, dvs först gör vi en grov skiss av klasserna för att sedan stegvis förfina dem och motivera elegantare implementationstekniker.

Ny definition av Person

Den nya definitionen av klassen blir:

```
Object subclass: #Person
    instanceVariableNames: 'name'
    classVariableNames: ' '
    poolDictionaries: ' '
    category: 'bok-1-person'
```

där vi har samlat ihop både för- och efternamn under den enda rubriken (instansvariabeln) name, som ska vara en instans av klassen Name.

En vanlig teknik för att på ett bekvämt sätt initiera en instansvariabel, och därmed se till att den blir av lämplig typ, är att utnyttja att alla variabler får värdet nil från början. Detta kan vara bra eftersom man vill befria användare av klassen från att ha specifik kännedom om klassens insida. Man kan också använda detta till att garantera att eventuell initiering utförs endast en gång.

Vi utnyttjar detta till att skriva en metod assureName som undersöker om instansvariabeln name är initierad, dvs skild från nil, och om så ej är fallet på lämpligt sätt initierar denna.

```
assureName
    name isNil
        ifTrue:
            [name := Name new.
            name christianNames: OrderedCollection new]
```

Det finns dock fortfarande en del saker som man kan anmärka på med denna lösning. Tex att Person måste ha kännedom om klassen Name, dvs veta att christianNames måste vara en OrderedCollection. Det kan också tänkas att en användare av Person vill använda en mer avancerad klass

för att hantera namn tex för att särbehandla tilltalsnamnet på ett bättre sätt.

```
name
    self assureName.
    ^name
```

Det kan tyckas onödigt och ineffektivt att meddelandet assureName skickas till det aktuella objektet varje gång som metoden name utförs. I avsnitt 4.4 kommer vi att visa lösningar på detta problem.

Vi skriver också om metoden christianName på följande sätt.

```
christianName
    ^self name christianName
```

Observera: Genom att använda metoden name istället för att instansvariabeln direkt så garanteras att name kommer vara av lämplig typ, dvs skild från nil.

Vi skriver också om metoden isValid och överlåter till name att avgöra om ett namn är riktigt istället.

```
isValid
    ^self name isValid
```

Vi kan skriva denna metod i Person trots att det ännu inte finns någon metod isValid i klassen Name. Vid kompilering gör Smalltalk endast en kontroll av om använda meddelanden finns som motsvarande metodnamn någonstans i systemet och inte om meddelandet har någon mening i det aktuella fallet.

Att vid kompileringstillfället undersöka om ett visst meddelande är meningsfullt är mer eller mindre omöjligt. Detta till stor del beroende av att Smalltalk är otypat (dynamiskt typning), dvs man behöver inte deklarera vilken typ av objekt variabler kan tilldelas. Även om något av de använda meddelandena inte tidigare existerar så ger Smalltalks interaktiva omgivning programmeraren chans till att ersätta dessa icke kända metodnamn, systemet föreslår tom liknande namn, fortsätta med behållande av aktuellt metodnamn eller att helt enkelt avbryta kompileringen.

I vårt fall finns det redan en metod med namnet isValid, bla den metod vi skriver har ju detta namn, så kompileringen kommer genomföras utan avbrott.

Name

Nu skriver vi isValid i klassen Name.

```
isValid
    ^self christianNamesIsValid  & self familyNameIsValid
```

Här har vi återigen använt oss av två nya hjälpmetoder som vi ännu inte har definierat. Den första, christianNamesIsValid, ska undersöka om förnamnen är instans av ett sekvensierbart objekt och om så är fallet undersöker vi också om storleken av detta objekt är minst ett. Metoden isSequenceable är definierad i klassen Object och förstås därmed av alla objekt. Från metoden i Object returneras alltid false men metoden är också omdefinierad i klassen SequenceableCollection och härifrån returneras som väntat alltid true. Eftersom även strängar är sekventierbara objekt så måste vi kontrollera så att christianNames inte är en sträng.

```
christianNamesIsValid
    ^self christianNames isSequenceable and: [
        self christianNames isString not
            and: [self christianNames size >= 1]]
```

Vi definierar att ett efternamn är korrekt om det är en instans av String eller någon av dess subklasser (testas med den unära metoden isString). Vi har också valt att använda det lata metoden and: som endast utför blocket som ges som argument om mottagaren är sann.

```
familyNameIsValid
    ^self familyName isString
```

Nu kan vi pröva det hela genom att tex skriva följande kodavsnitt i ett arbetsfönster, måla över och exekvera mha menyalternativet **do it**.

```
| person outString |
person := Person new.
person christianName: 'Albert'.
person isValid
        ifTrue: [outString := 'Name: ', person christianName, ' ',
            person surname]
        ifFalse: [outString := 'Ej giltigt personobjekt'].
Transcript show: outString.
Transcript cr.
Transcript print: person name christianNames.
Transcript endEntry
```

I det ovan beskrivna fallet kommer person isValid returnera false, då vi ännu inte har givet person något efternamn. Därmed kommer false-delen av det efterföljande villkoret utföras och outString tilldelas ett 'felmeddelande'. Slutligen skrivs outString och förnamnen ut i utskriftsfönstret.

Sammanfattning

Termer

Rotklass klassen högst upp i arvsträdet.

Primitiv metod en metod som utförs direkt av den virtuella maskinen.

Litteral konstant objekt som kan skapas direkt med speciella syntaktiska konstruktioner.

Kaskadmeddelande för att på ett enkelt sätt skicka flera meddelanden till samma objekt.

Behållare ett objekt som består av flera andra objekt.

Programmeringsmetodik

Metoder *omdefinieras* i subklasser. Vilken metod som används avgörs av mottagande objekts klasstillhörighet.

Klasser ska vara så lite beroende av andra klasser som möjligt. Man ska utifrån inte behöva veta hur klassen ser ut inuti.

Smalltalk

Smalltalk använder *referenstilldelning*.

Tre typer av meddelanden: u*nära, binära* och *nyckelordsmeddelanden*.

Prioritetsordningen är

1. *unära*
2. *binära*
3. *nyckelordsmeddelanden*

Vid lika prioritet utförs meddelandena från vänster till höger. Parenteser kan användas för att ändra ordningen.

Test av klasstillhörighet kan göras med: class, species, isMemberOf:, subclasses, superclass, isString, isInteger.

Test av mottagningsförmåga kan göras med respondsToArithmetic, isSequenceable.

Litterala konstanter skrivs:

Tal, som man är van vid, fast olika baser kan användas.

Tecken, med dollartecken framför ($)

Strängar, inom apostrofer (')

Symboler, med nummertecken (#)

Vektorer, med nummertecken och runda parenteser (#())

Byte-vektorer, med nummertecken och hakparenteser (#[])

Lika eller olika värden testas med = respektive ~=

Identiska eller olika identitet testas med == respektive ~~

Är objektet nil eller inte kan testas med isNil eller notNil

Kopiering av objekt kan göras med

shallowCopy dvs grund kopiering där instansvariabler delas mellan original och kopia

copy där metoden *postCopy* definierar vad som skall hända utöver shallowCopy.

self en speciell variabel som refererar till det objekt som utför en metod, dvs mottagaren av motsvarande meddelande.

super en speciell variabel som precis som self refererar till det objekt som utför en viss metod. Indikerar att metod vid meddelande till objektet ska börja sökas i klasser högre upp i klasshierarkin än var den aktuella metoden är definierad.

thisContext refererar aktuell exekveringsomgivning.

true speciell variabel som representerar konstanten sant.

false speciell variabel som representerar konstanten falskt.

nil anger det värde som en variabel har om inget speciellt objekt har tilldelats det. Alla variabler har detta värde från början.

Övningar

3.1 Vilken klass tillhör den temporära variabeln efter det att koden i respektive deluppgift i a-c nedan har utförts?

a) a := 4 / 12.

b) b := nil

c) c

3.2 Tillhör

a) objektet 2/3 klassen Fraction?

b) variabeln x givet på följande sätt x := 'en sträng' klassen Float?

3.3 Vilken klass är superklass till

a) klassen Fraction

b) klassen för 4

3.4 Vilka är de närmaste subklasserna till

a) Number

b) superklassen till vektorn #('test')

3.5 På sidan 88 finns metoderna christianName och familyName:. Skriv motsvarande christianName: och familyName.

3.6 Varför kan vi inte i assureName på sidan 89 skriva self name istället för bara name?

3.7 Finn genom intervallhalvering nollstället till *sin* i intervallet (-1,1). Algoritm, antag f(a) < 0 och 0 < f(b):

```
solve(f, a, b)
{
  m = (a+b)/2;
  if (abs(f(m)) < eps)
    return m
  else (if f(m) < 0)
    return solve(f, m, b);
  else
    return solve(f, a, m);
}
```

3.8 Vad blir resultatet av följande?

a) 3 * 4 factorial + 5 max: 7 + 5 factorial

b) 3 * 4 factorial; + 5; max: 7; + 5 factorial

c) 3 * 4 factorial; + (5 max: 7; + 5 factorial)

4 Klasser, arv och metoder

Detta kapitel besvarar bland annat följande frågor:

- Hur använder man arv?
- Hur utnyttjar man det som redan är gjort i superklassens metod när man omdefinierar en metod i en subklass
- Hur kommer man åt en metod från en subklass även om den aktuella metoden är omdefinierad i denna subklass.
- Vad är en klassvariabel?
- Vad är en poolvariabel?
- Vad är en metaklass?

4.1 Instansmetoder

Hittills har vi sett många exempel på instansmetoder, dvs metoder som beskriver beteendet för instanserna av en viss klass.

Returvärde från metod

Om ingenting annat anges så returneras en referens till det mottagande objektet som *svar* från en metod. Detta kan förändras genom att man i metoden anger vad som ska returneras. I så fall skrivs en stiliserad uppåtpil (^) följt av det objekt, eller ett uttryck som genererar det, som ska returneras.

Följande metod, som tex kan definieras i klassen Object, är ett exempel på en metod från vilken ett annat objekt än mottagaren returneras:

```
doSomething
    ^2 ** 3
```

I det här fallet returneras resultatet av metodens enda sats, som beräknar 2^3 och ger heltalet 8 som resultat.

```
'test' doSomething                    543 doSomething
⇒8                                    ⇒8
```

Om vi inte hade använt retursymbolen hade meddelandesändningen istället resulterat i en referens till det mottagande objektet. Dvs i det första fallet strängen 'test' och i det andra fallet heltalet 543. Ett minimalt exempel på en metod som returnerar en referens till det mottagande objektet är metoden yourself, som är definierad i klassen Object. Den ser ut som följer:

```
yourself
```

Detta är enligt regeln om att self returneras om inget annat anges ekvivalent med:

```
yourself
    ^self
```

Metoden yourself, som egentligen inte gör någonting, brukar användas i samband med kaskadmeddelanden (se avsnitt 4.7) och utnyttjas om man vill tvinga fram att en pekare till mottagande objekt returneras (oavsett vad det sista meddelandet i "kaskaden" returnerar). Vi kommer se exempel på dess användning på sidan 247.

Placering av retursymbolen

En viktig fråga är hur värden returneras från en metod. Regeln är att retursymbolen kan placeras:

före metodens sista sats

ett exempel på detta är metoden isAlphabetic i klassen Character, som svarar true om mottagaren är ett tecken i det engelska alfabetet, annars blir svaret false.

```
isAlphabetic
    | code |
    code := self asInteger.
    ^code <= 255
        ifTrue: [((ClassificationTable at: code + 1)
                bitAnd: AlphabeticMask) ~= 0]
        ifFalse: [false]
```

i block som används av metoden

i ett sådant block ska retursymbolen antingen vara placerad före dess sista sats eller i ett inre block.

Metoden detect:ifNone: i den abstrakta klassen Collection får illustrera detta.

```
detect: aBlock ifNone: exceptionBlock
    self do: [:each | (aBlock value: each) ifTrue: [^each]].
    ^exceptionBlock value
```

Metoden applicerar aBlock på vart och ett av det mottagande behållarobjektets element och returnerar det första elementet som resulterar i att aBlock returnerar true. Om alla element resulterar i false så utförs det andra blocket, exceptionBlock.

Metoder för att läsa respektive ändra attribut

I Smalltalk är alla attribut, dvs instansvariabler, klassvariabler och poolvariabler skyddade från åtkomst utanför objekten där de är definierade. Om man vill att andra objekt ska kunna komma åt eller förändra värdet av en instansvariabel så måste objektets klass erbjuda metoder för detta.

En klassvariabel delas däremot mellan den klass som definierat den, dess subklasser samt alla instanser av dessa. Oavsett hur många instanser av klassen som skapas, finns det bara en instans av själva klassvariabeln. En poolvariabel fungerar, i fråga om åtkomst, analogt med en klassvariabel fast med den skillnaden att klasser och deras instanser från flera grenar av arvsträdet kan dela dem.

Variabeltyp:	instans	klass	pool
Av instans	x	x	x
Alla instanser av klassen		x	x
Klass		x	x
Klasser ur olika arvsträd			x

Figur 4.1 Åtkomlighet för variabel

Vi definierar följande enkla klass som har Object som superklass och endast deklarerar ett attribut, attribut, i form av en instansvariabel.

```
Object subclass: #Dummy
    instanceVariableNames: 'attribut'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Godtycklig'
```

Läsa attribut

För att möjliggöra tillgång av ett visst objekts attribut även utanför det aktuella objektet så måste metoder som returnerar dessa skrivas i klassen. Sådana metoder kallas för *inspektorer*.

Följande metod returnerar variabeln attribut.

```
åtkomstMetodNamn
    ^attribut
```

Metoder för att läsa attribut brukar ha samma namn som attributet, vilket gör att föregående metod istället borde se ut som följer:

```
attribut
    ^attribut
```

Skriva eller omdefiniera attribut

Om andra objekt än det egna ska ha möjlighet att tilldela attribut nya värden så måste objektets klass tillhandahålla metoder för detta. Sådana metoder kallas för *mutatorer.*

```
uppdateringsMetod: nyttVärde
    attribut := nyttVärde
```

Metoden för att ändra värdet av ett attribut brukar ofta ha samma namn som det aktuella attributet (om man bortser från kolonet och den formella parametern). Detta ger följande utseende på mutatorn för attributet attribut:

```
attribut: nyttVärde
    attribut := nyttVärde
```

Argument och parametrar

En binär eller nyckelordsmetod har ett eller flera argument alla representerade av formella parametrar. En parameter till en metod kan *aldrig* tilldelas ett nytt värde.

```
test: parameter
    parameter := 'Detta går ej!'
```

Men attributen representerar ju också objekt som tillhör klasser, vilket innebär att det är möjligt att skicka meddelanden till dessa argument, även om de har sidoeffekter. På så sätt går det t ex att skicka en vektor som argument till ett meddelande och förändra dess innehåll.

Så låt oss helt hypotetiskt anta att vi skriver följande metod i en klass Mother:

```
timeStamp: aCollection
    aCollection add: Time now
```

Metoden antar helt enkelt att argumentet är en behållare som förstår meddelandet add: och använder detta för att tillfoga aktuellt klockslag i till detta.

Nu kan vi utföra följande test:

```
| collection mother |
collection := #(123 abc) asOrderedCollection.
mother := Mother new.
mother timeStamp: collection.
collection
```

Som också följdriktigt ger variabeln collection det nya värdet:

⟹*OrderedCollection (123 #abc 3:18:47 pm)*

4.2 Klassmetoder

Klassmetoder är väsentligen ekvivalenta med instansmetoder. Skillnaden som finns ligger i huvudsak i vilket objekt som är mottagare av meddelandet. Instansmetoder kräver att det finns en instans som mottagare till meddelandet, klassmetoder använder klassen som mottagare. I grund och botten kan vi säga att det hela baseras på att allt i Smalltalk är objekt, inklusive klasser! Då ett objekts beteende i huvudsak definieras av dess metoder och vi just kommit fram till att klasserna också är objekt så medför detta att också klassernas beteende definieras av dess metoder. Dessa klassmetoder konstrueras på precis samma sätt som instansmetoderna.

Vi har redan tidigare ofta använt oss av metoder som är definierade som klassmetoder, utan att egentligen poängtera detta. Som exempel kan vi nämna att i följande Smalltalk-utryck används enbart klassmetoder.

```
Person new.
Object new.
Time now.
Date today.
Array new: 5.
OrderedCollection with: 'Klassen' with: 'är mottagare' with: 'av meddelandet!'
```

Som framgår är klassmetoder helt enkelt metoder som definierar beteendet för meddelanden till klasser.

En klass är en instans av en klass

Anledningen till att Smalltalks klassmetoder inte skiljer sig från dess instansmetoder är klasserna också är objekt som i sin tur är instanser av klasser.

Metaklasser

Objekt vars instanser i sin tur är klasser kallas för *metaklasser*. I Smalltalk har metaklasser inget eget namn utan refereras via meddelandet class till dess enda instans, dvs klassen.

Alla klasser i Smalltalk är instanser av metaklasser. Detta medför att Object är instans av Object class, OrderedCollection instans av OrderedCollection class, Float instans av Float class, osv.

En metaklass skapas alltid automatiskt då en ny klass definieras. En sådan metaklass blir i sin tur instans av klassen Metaclass. När vi skapade klassen Dummy skapar systemet också automatiskt dess metaklass Dummy class.

För att undersöka eller förändra en metaklass med hjälp av browsern trycker vi helt enkelt på knappen märkt **class** nederst i klasslistan. Sedan fungerar allt som vanligt, dvs som då vi förändrar klassen när knappen **instance** är nedtryckt.

Arvsträden för en klass och dess metaklass följer varandras hierarkier så att tex Integer's superklass är Number och Integer class' superklass är Number class. Dessa hierarkier följer varandra ända upp till och med Object och Object class. Här slutar klasshierarkin för de vanliga klasserna med klassen Object.

I figur 4.2 är den ovan beskrivna hierarkin avbildad och i det högra trädet ser vi bl a att klassen Class är superklass till Object class. Sedan fortsätter även det högra trädet med ”vanliga” klasser ända tills dess att vi också här når den ultimata superklassen Object! Detta är faktiskt samma klass Object som i den vänstra klasshierarkin, se kapitel 6.

Klass	**Metaklass**
	Object Behavior ClassDescription Class
Object Magnitude ArithmeticValue Number Integer	Object class Magnitude class ArithmeticValue class Number class Integer class

Figur 4.2 Klass- och metaklasshierarki

Klasserna Class, ClassDescription och Behavior som är med i figuren samt klassen Metaclass beskriver struktur och beteende för alla Smalltalks

klasser som sådana. De beskriver egenskaperna som skiljer klasser från vanliga objekt, t ex vilka metoder som finns och hur de är definierade

Exempelvis så innehåller klassen Behavior instansvariblerna superclass, subclasses och methodDict som pekar ut aktuell klass superklass, dess subklasser respektive associerar meddelandenamn med metoder.

Klassen Behavior definierar också metoderna new, basicNew, och new: som instansmetoder. Detta medför ju att dessa meddelanden förstås av alla klasser, då dess metaklasser alla är subklasser till Behavior, se figur 4.2 ovan.

Vi kan också nämna att beskrivning och hantering av en klass instansvariabler samt organisation av dess olika metoder (dvs deras tillhörighet till metodkategorier) sker i klassen ClassDescription.

Klassen Class definierar olika metoder för att konstruera nya subklasser till en existerande klass, tex de olika varianterna på meddelandet:

```
subclass:
    instanceVariableNames:
    classVariableNames:
    poolDictionaries:
    category:
```

som ju är precis det meddelande som visar sig i browserns textfönster vid definition av klasser, fast då med en mottagare och argument redan ifyllda.

Listan av intressanta metoder i klasserna som hanterar klasser kan göras lång och vi återkommer till några av dem bland exemplen i slutet av kapitlet. Vi kommer också beskriva metaklasser i mer detalj i kapitel 6. För den intresserade läsaren påminner vi om att som vanligt ta reda på mer om systemet genom att använda en browser. I detta fall med fokus på klasskategorin Kernel-Classes i vilken alla klasser som hanterar klasser befinner sig.

Metoder definierade i metaklassen

Precis som klassen beskriver de metoder som är definierade för instanser av klassen beskriver metaklassen de metoder som är definierade för klassen, som också är metaklassens enda instans.

Detta gör det enkelt att skriva metoder som är avsedda för klassen. Det är bara att använda browsern och skriva metoderna som förut. En väsentlig skillnad är att det endast finns en enda instans av klassen och en av dess huvuduppgifter är att konstruera nya instanser ur den klass den beskriver.

Typexempel

Som vi nämnt används meddelandena new och new: för att konstruera instanser av en viss klass, där det första är det vanliga sättet att konstruera instanser utan speciell initial information och det senare för att konstruera instanser av variabel typ, tex Array, med viss storlek.

```
mängd := Set new.
vektor := Array new: 25.
```

Ibland vet man redan då objektet skapas vissa attributs värde. Därför kan det vara praktiskt med klassmetoder som initierar dessa attribut.

```
tvåkomponentsVektor := Array with: 'A Little Smalltalk' with: 'Timothy Budd'.
inköpslista := OrderedCollection withAll: #('kaffe' 'smör' 'bröd' 'mjölk' 'tomater')
```

För vissa klasser kan instansieringen göras med hjälp av klassmetoder.

```
dagensDatum := Date today.
klockanÄrNu := Time now
```

Klassmetoder som program

Förutom för att skapa nya instanser av klasser brukar klassmetoder användas till att öppna användargränssnitt mot något, troligen nyskapat, objekt. Bland klassmetoderna finner man också ofta metoder som snarare kan betraktas som program i mer konventionell mening. Exempel på detta kan man hitta i hela systemet där klassmetoder ofta används för att starta exempelkod eller grafiskta verktyg.

Det är bra att ta för vana att skriva exempelkod som klassmetoder istället för i ett arbetsfönster. Fördelen är att man får exemplen samlade och enkelt kan skriva ut dem på fil samt att browserns kodformateringsmöjligheter kan användas.

Konstanter kan också lämpligen definieras som klassmetoder däregenom får man god kontroll på var i systemet de är definierade och de kan också enkelt lagras på fil tillsammans med klassbeskrivningen.

Programexempel

Några typiska exempel på klassmetoder som kan ses som program är följande tre som är hämtade från VisualWorks.

```
Browser open.
Bezier fromUser.
Printer example2
```

Genom att söka bland klassmetoderna i systemet hittar du fler exempel i alla Smalltalksystem.

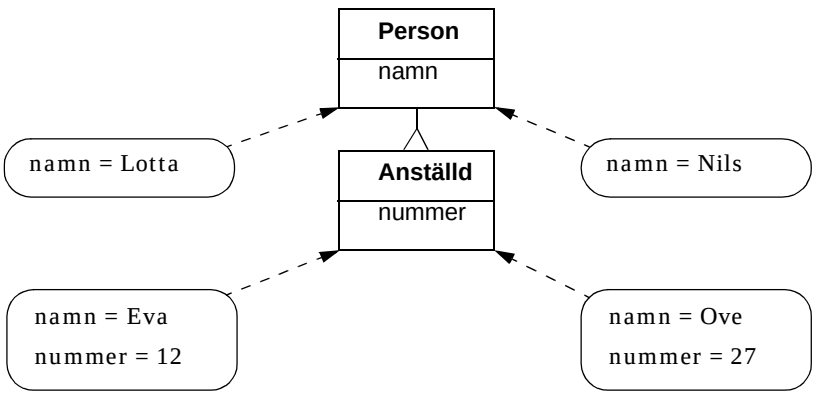
4.3 Variabler i klasser

En klass kan definiera en mängd olika attribut. Hittills har vi endast sett exempel på instansvariabler, dvs variabler som är unika och lokala för varje instans av klassen. I det här avsnittet ska vi lite mer i detalj diskutera de olika variabeltyperna som förekommer i en klassdefinition nämligen *instansvariabler*, *klassvariabler* och *poolvariabler*. Vi kommer också diskutera hur *klassinstansvariabler* kan deklareras och användas samt slutligen beskriva *temporära variabler* (lokala variabler) i metoder och block.

Instansvariabler

Vi har tidigare använt begreppet *instans* för att tala om ett objekt ur en viss klass. På samma sätt används termen *instansvariabel* för att referera till attribut som är relaterat till en viss instans. Klassen definierar instansvariablernas förekomst genom att de deklareras tillsammans med klassdefinitionen. Instansvariablerna som är definierade i en viss klass är sedan lokala och har unika värden för varje instans av klassen. En instansvariabel får som alla andra objekt värdet nil från början.

En klass ärver alla instansvariabler från alla sina superklasser och kan sedan utöka denna mängd av instansvariabler vilka i sin tur kommer ärvas av den aktuella klassens eventulla subklasser. Det går inte att ”göra sig av med” en instansvariabel i en subklass om den är definierad i någon superklass.



Figur 4.3 Klasshierarki med instansvariabler

En instansvariabel inleds med en gemen (liten) bokstav. Om instansvariabelns namn är uppbyggt av flera sammanslagna ord så skrivs den, som brukligt i Smalltalk, med varje delord inlett med en versal. Förutom dessa versaler för delord skrivs resten av instansvariabeln med gemena bokstäver.

En definition av klasserna i figur 4.3 ser ut på följande sätt:

```
Object subclass: #Person
    instanceVariableNames: 'namn '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Personer-och-anställda'

Person subclass: #Anställd
    instanceVariableNames: 'nummer '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Personer-och-anställda'
```

Vi förutsätter att metoderna namn: i klassen Person och nummer: i klassen Anställd är skrivna och prövar det hela.

```
| p1 p2 a1  a2 |
p1 := Person new.
p2 := Person new.
p1 namn: 'Lotta'.                    ge instansvaribeln namn dess värde
p2 namn: 'Nils'.

a1 := Anställd new.
a1 namn: 'Eva'.
a1 nummer: 12.          objekt ur klassen anställd har även nummer definierat
a2 := Anställd new.
a2 namn: 'Ove'.
a2 nummer: 27.
```

Temporära variabler

Temporära varibaler kan användas för att mellanlagra lokala värden i metoder eller block.

En temporär variabel kan returneras och därmed få objektet som den pekar ut att leva vidare. Detta illustrerar vi med en temporär variabel i ett block.

```
| ytte |
block := [| inre |
    inre := 'Den inre variabeln tilldelas denna sträng'.
    inre].
yttre := block value
```

⟹'Den inre variabeln tilldelas denna sträng'

I en metod kan vi använda temporära variabler för att mellanlagra delresultat. Ett exempel på detta är följande metod i klassen Point, som avrundar en punkt till ett rutnät vars finkornighet är given av argumentet.

```
grid: aPoint
    | newX newY |
    aPoint x = 0
        ifTrue:[newX := 0]
        ifFalse:[newX := x roundTo: aPoint x].
    aPoint y = 0
        ifTrue:[newY := 0]
        ifFalse:[newY := y roundTo: aPoint y].
    ^newX @ newY
```

Pseudo- eller speciella variabler

Det finns ett antal olika *pseudo-* eller *speciella variabler*. Av dessa är det speciellt viktigt att känna till och behärska variablerna *self* och *super*.

Precis som vi inte kan tilldela ett nytt värde till en formell parameter kan vi inte heller tilldela en pseudovariabel ett nytt värde.

variabeln self

Varibeln self ger oss möjlighet att inuti en klass metoder referera till den instans som är mottagare av meddelandet. Med dess hjälp kan vi alltså skicka meddelanden till den ”exekverande” instansen men också använda objektet som argument till andra metoder. Slutligen kan vi använda self för att returnera referenser till det mottagande objektet från dess metoder.

```
metod
    self meddelandeFrån: self          meddelandet skickas med aktuellt objekt
                                       som både mottagare och argument
```

Nu skriver vi en metod info i klassen Person. Metoden returnerar en ledtext följd av den aktuella instansens namn. För att innifrån metoden använda den egna metoden namn skickas motsvarande meddelande till self.

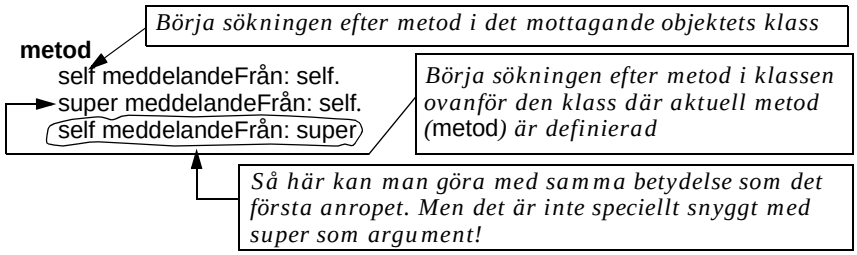
```
info
    ^'Namn: ', self namn                    strängar konkateneras med komma.
```

Nu prövar vi det hela med den temporära variabeln p1, instans av Person, från exemplet på sidan 104.

```
p1 info
⇒'Namn: Lotta'
```

Variabeln super

Variabeln super ger också en referens till aktuell instans. Det som skiljer super från self är att metoduppslagning börjar en nivå upp i förhållande till var aktuell metod är definierad. Men förutom skillnaden vid metoduppslagning, vilket vi utreder senare i avsnitt 4.7, är self och super *identiska*!



Figur 4.4 self och super

Sammanfattningsvis, kom ihåg att

self ≡ super

och att super endast styr *metoduppslagningen* genom att indikera att den ska börja i klassen närmast ovanför den klass där den aktuella metoden är definierad (normalt börjar ju sökningen i det mottagande objektets klass).

Vi konstruerar en metod info i klassen Anställd. Vi använder superklassens info och tillfogar nummer till dess resultat.

```
info
    ^super info, '
    Anställningsnummer: ', self nummer printString      printString omvandlar
                                                                till sträng
```

Nu prövar vi också detta på den temporära variabeln a2, som är instans av Anställd, från kodexemplet på sidan 104.

```
a2 info
```

⇒

'Namn: Ove
Anställningsnummer: 27'

Pseudovariabeln thisContext

Det finns också en pseudovariabel, thisContext, som är starkt knuten till beskrivningen av metoder och exekvering av dem.

Med dess hjälp kan programmeraren få tillgång till en viss metods aktuella exekveringsomgivning. Den intresserade läsaren hänvisas till manualer och boken Interaktionprogrammering i Smalltalk.

Formella parametrar

Alla meddelanden utom de unära har argument. Dessa argument representeras av *formella parametrar* i metoderna.

Det går inte att tilldela ett nytt värde till en formell parameter men det går att skicka meddelanden till den. Ett sådant meddelande kan ha sidoeffekter, så viss försiktighet är på sin plats.

Namnet på en formell parameter består av bokstäver ur det vanliga alfabetet. Den första bokstaven ska vara gemen.

Man bör ge de formella parametrarna ett så förklarande namn som möjligt och om typen är viktig bör detta också indikeras. Exempelvis om argumentet alltid ska vara en instans av Point bör den formella parametern heta aPoint eller om den alltid ska vara en ordnad lista bör namnet vara aSequenceableCollection.

Nu skriver vi metoden namn:nummer: i klassen Anställd med konventionen att de formella parametrarna ska vara förklarande.

```
namn: ettNamn nummer: ettHeltal
    self namn: ettNamn.
    self nummer: ettHeltal
```

Klassvariabler

En *klassvariabel* delas mellan alla instanser av en klass och klassen själv. Ändras klassvariabeln av någon av dessa objekt så kommer detta automatiskt ändras för alla objekt som delar denna variabel. Orsaken är att en klassvariabel alltid är en pekare till ett, och endast ett, gemensamt objekt för de som delar den aktuella klassvariabeln.

Att klassen också är med om att dela klassvariabeln medför att den kan ha ett värde även om det inte finns några instanser av klassen eller någon av dess subklasser.

Klassvariabler namnges med en versal som första bokstav.

Exempel med klassvariabel och initiering av nya instansers attribut med klassvariabeln som skönsvärde

```
Object subclass: #Dummy
    instanceVariableNames: 'attribut'
    classVariableNames: 'DefaultValue'
    poolDictionaries: ''
    category: 'Godtycklig'
```

Vi skriver om instansmetoden initialize så att den använder klassens skönsvärde istället.

```
initialize
    attribut := DefaultValue
```

Förutom att en klassvariabel delas mellan klassen och instanser av densamma så delas den också av alla subklasser och deras respektive instanser.

Nu definierar vi en klassvariabel ID avsedd att automatiskt generera nya anställningsnummer för instanserna av klassen Anställd.

```
Person subclass: #Anställd
    instanceVariableNames: 'nummer '
    classVariableNames: 'ID '
    poolDictionaries: ''
    category: 'Personer-och-anställda'
```

Vi skriver också följande klassmetod för att kunna initiera klassvariabeln.

```
initialize
    ID isNil ifTrue: [ID := 0]
```

Vi initierar det hela genom att utföra följande:

```
Anställd initialize
```

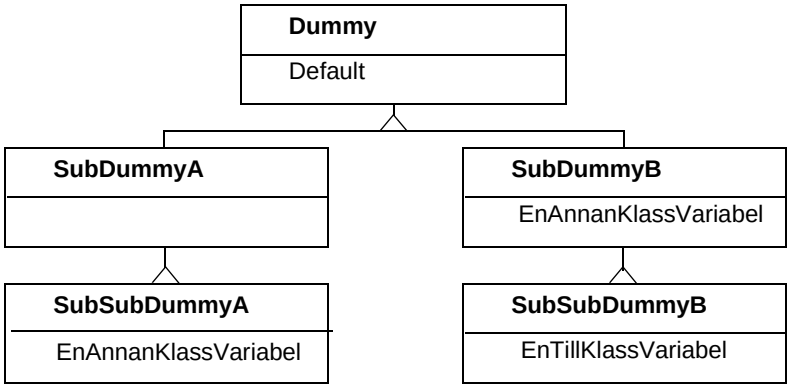
Nu definierar vi om metoden name: så att den automatiskt genererar nya idnummer för en instans vid det tillfället den anställda namnges.

```
namn: enSträng
    self namn: enSträng nummer: (ID := ID + 1)
```

Nu bör inte metoden name:nummer: vara åtkomlig utifrån längre utan den flyttas till kategorin private för att markera detta.

Flera klassvariabler samtidigt

En klass kan samtidigt ha flera olika klassvariabler, eventuellt ärvda. Två klassvariabler kan inte ha samma namn om de samtidigt är definierade för en gemensam klass. Däremot kan klassvariabler med samma namn användas i olika grenar av arvsträdet.



Figur 4.5 Klasshierarki med klassvariabler

I figur 4.5 har vi definierat en enkel klasshierarki alla med klassen Dummy som gemensam superklass. I Dummy finns klassvariabeln Default och i tre av subklasserna definierar vi nya klassvariabler.

Alla klasser och deras instanser har tillgång till klassvariabeln Default. Om tex en instans av SubDummyA ändrar Defaults värde ändras det också för alla de andra fyra klasserna och deras instanser. Däremot finns det två olika klassvariabler med namnet EnAnnanKlassVariabel, en definierad i klassen SubDummyB och en i klassen SubSubDummyA. Så om klassen SubDummyB dess subklasser eller någon ev deras instanser ändrar på variabeln EnAnnanKlassVariabel så påverkar detta inte grenen SubSubDummyA då dess klassvariabel endast är densamma till namnet.

Ett exempel på en klassvariabel som beroende av klasstillhörighet antar olika värden är Pi som är ett flyttal i enkel precision i klassen Float men ett flyttal i dubbel precision i klassen Double.

Exempel: Konvertering mellan olika temperaturgradskalor

Det välkända sambandet mellan grader Kelvin (K) och Celsius (C) kan utryckas med följande formel: $K = C + 273.15$.

Vi ska nu konstruera två klasser som var och en beskriver en instans av den respektive gradskalan.

För enkelhets skull ser vi celsius som den universella gradskalan och genererar alltid nya objekt ur båda klasserna genom att ange celsiusgrader.

Klasserna har ett attribut numericValue som anger temperaturen i den aktuella gradskalan.

Slutligen ska instanser av Celsius via en unär metod asKelvin kunna ge oss sin motsvarighet uttryckt i kelvin och vice versa, fast då heter konverteringsmetoden asCelsius. Klasserna ska förstå båda dessa konverteringsmeddelanden och reagera på lämpligt sätt.

Lösning:

Vi börjar med klassen Celsius, som är ganska rakt på.

```
Object subclass: #Celsius
    instanceVariableNames: 'numericValue '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Temperatur'
```

De olika metoderna för att sätta respektive läsa attribut placerar vi lämpligen i en metodkategori accessing.

```
fromCelsiusValue: celsiusValue
    self numericValue: celsiusValue

numericValue
    ^numericValue

numericValue: newValue
    numericValue := newValue
```

Metoderna för att konvertera objekt brukar ofta placeras i en kategori med namnet converting. Vi följer denna konvention.

```
asCelsius
    "Ingen konvertering behövs, så vi returnerar helt enkelt self"
    ^self

asKelvin
    "Returnera min motsvarighet i kelvingrader"
    | kelvin |          Vi kan använda klassen Kelvin i metodbeskrivningen
    kelvin := Kelvin new.          trots att den inte är definierad ännu!
    kelvin fromCelsiusValue: self numericValue.
    ^kelvin
```

I klassen Kelvin väljer vi att använda en klassvariabel, ToCelsiusDiff, för att utrycka sambandet mellan Kelvin och Celsius.

```
Object subclass: #Kelvin
    instanceVariableNames: 'numericValue '
    classVariableNames: 'ToCelsiusDiff '
    poolDictionaries: ''
    category: 'Temperatur'
```

Metoderna numericValue och numericValue: blir helt ekvivalenta med deras motsvarigheter i Celsius. Däremot använder vi klassvariabeln för att omvandla de givna celsiusgraderna till Kelvin i fromCelsiusValue:.

```
accessing

fromCelsiusValue: celsiusValue
    "Argumentet som är givet i celsius omvandlas till kelvin och sparas"
    self numericValue: celsiusValue + ToCelsiusDiff

numericValue
    ^numericValue

numericValue: newValue
    numericValue := newValue
```

Konvereringsmetoderna blir helt analoga med motsvarigheterna i Celsius.

```
converting

asCelsius
    "Returnera min motsvarighet i celsius"
    | celsius |
    celsius := Celsius new.
    celsius fromCelsiusValue: self numericValue - ToCelsiusDiff.
    ^celsius

asKelvin
    "Min motsvarighet i Kelvin. Trivialt!"
    ^self
```

Vi får inte glömma att initiera klassvariabeln i klassen Kelvin. Då vi inte har någon metod för detta ännu så kan vi antingen initiera den direkt i en browser, dvs göra tilldelning och använd menyalternativet **do it**, eller använda ett inspektfönster, dvs inspektera klassen, leta reda på variabeln och göra följande tilldelning:

```
ToCelsiusDiff := 273.15
```

Vi provar det hela:

```
| c k |
    c := Celsius new.
    c fromCelsiusValue: 10.

    k := Kelvin new.
    k fromCelsiusValue: 10.

    c asKelvin numericValue = k numericValue
            and: [c numericValue = k asCelsius numericValue]
```

⇒*true*

Exempel: Temperatur i Fahrenheit

Vi utökar exemplet med en klass för att hantera temperatur i Fahrenheit. Relationen mellan Fahrenheit (F) och Celsius (C) kan uttryckas på följande sätt: $C = \frac{F - 32}{5} \cdot 9$.

Lösning:

Vi använder två olika klassvariabler för att utrycka sambandet, en för att konvertera från celsius till fahrenheit och en för att konvertera från fahrenheit till celsius. Då sambandet inte är lika enkelt som i kelvinfallet duger det inte med konstanter utan vi använder ett block. Blocket definieras med en parameter som ska vara ett gradtal uttryckt i celsius respektive fahrenheit.

Eftersom koden är så pass lik den i Celsius och Kelvin är fler kommentarer överflödiga och vi listar upp klassen i sin helhet på en gång.

```
Object subclass: #Fahrenheit
    instanceVariableNames: 'numericValue '
    classVariableNames: 'FromCelsius ToCelsius '
    poolDictionaries: ''
    category: 'Temperatur'

accessing

fromCelsiusValue: celsiusValue
    "Konvertera celsiusValue till min representation"
    self numericValue: (FromCelsius value: celsiusValue)

numericValue
    ^numericValue

numericValue: newValue
    numericValue := newValue

converting

asCelsius
    | celsius |
    celsius := Celsius new.
    celsius fromCelsiusValue: (ToCelsius value: self numericValue).
    ^celsius

asFahrenheit
    ^self


asKelvin
    ^self asCelsius asKelvin
```

Klassvariablerna initieras på motsvarande sätt som Kelvin's klassvariabler:

```
ToCelsius := [:f | f - 32 * 5 / 9].
FromCelsius := [:c | c * 9 / 5 + 32]
```

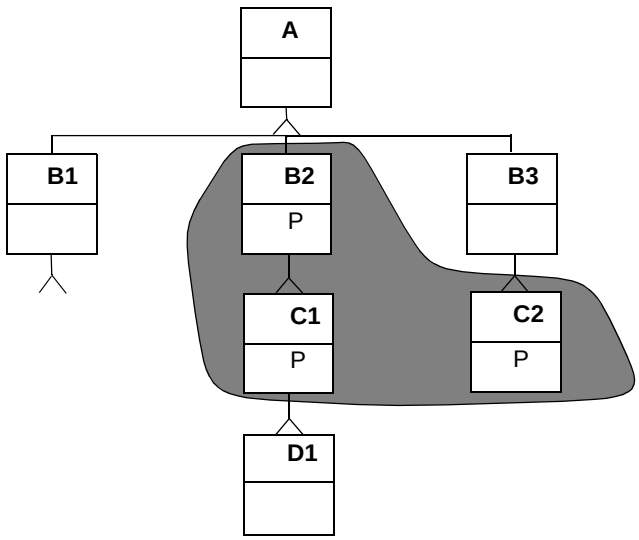
Nu måste vi också lägga till konverteringsmetoder för att konvertera till Fahrenheit.

```
Kelvin>>asFahrenheit
    ^self asCelsius asFahrenheit

Celsius>>asFahrenheit
    | fahrenheit |
    fahrenheit := Fahrenheit new.
    fahrenheit fromCelsiusValue: self numericValue.
    ^fahrenheit
```

Det hela kan testas på precis samma sätt som då vi testade Celsius och Kelvin tidigare.

Poolvariabler

Den sista typen av variabler som deklareras i klassdefinitionshuvudet är de så kallade *poolvariblerna*. En poolvariabel ligger i ett Dictionary, dvs en slags uppslagsbok, som kan delas mellan olika klasser. Här kallar vi katalogen för poolvariabelkatalogen. Om vi jämför med klassvariabler, som delas mellan alla instanser och deras klasser i den hierarki där den är deklarerad, så kan vi löst säga att en poolvariabel istället delas på tvären mellan klasser som inte nödvändigtvis behöver tillhöra samma gren av arvsträdet. I figur 4.6 illustreras detta genom att poolvariabelkatalogen P är definierad. De klasser som har tillgång till poolvariabelkatalogen är inringade och har en gemensam mörkare bakgrund.



Figur 4.6 En poolvariabelkatalog kan gå på tvären i klasshierarkin

En finess med att använda en poolvariabel, eller snarare poolvariabelkatalogen, är att man i koden direkt kan använda dess nycklar som variabler. Värdena för dessa variabler ges av det värde som för närvarande (dvs vid exekveringtillfället) är lagrat i poolen för aktuell nyckel. Observera att det inte går att använda en instansvariabel med samma namn som någon poolvariabel.

Observera att C1 också deklarerar poolvariabelkatalogen trots att dess superklass B2 redan har den deklarerad.

Vi anser att man bör vara försiktig med globala variabler och dit räknar vi poolvariabler. En orsak är att det delvis bryter mot vissa objektorienterade principer. En annan viktik orsak till att använda poolvariabler sparsamt är att de kan vara besvärliga att initiera på ett riktigt sätt då källkod flyttas.

Exempel: Att hantera olika språk i användargränssnitt

Poolvariabel som används för att enkelt byta mellan olika språk i ledtexter

För att inte få problem med deklarationen av poolvariabelkatalogen så ser vi först till att en global variabel med önskat namn konstrueras.

```
(Smalltalk includesKey: #PoolUserLanguageBinding)
    ifFalse:
        [Smalltalk at: #PoolUserLanguageBinding put: PoolDictionary new]
```

Sedan konstruerar vi en klass som beskriver ledtexter på engelska.

```
Object subclass: #PoolUserEnglish
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: 'PoolUserLanguageBinding '
    category: 'Test-Pool'
```

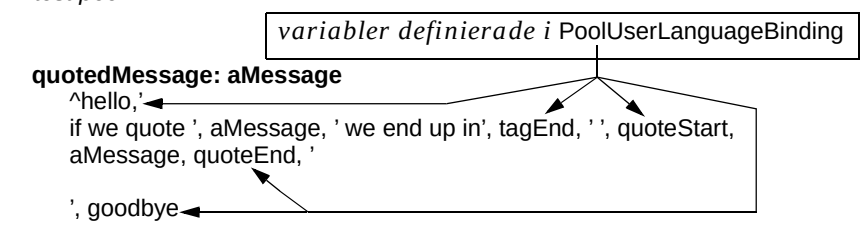
Vi skriver ett antal klassmetoder avsedda att användas för att både initiera och binda poolvariabeln till aktuella värden.

initialize-pool

```
initializePool
    (Smalltalk includesKey: #PoolUserLanguageBinding)
        ifFalse:
            [Smalltalk at: #PoolUserLanguageBinding put: PoolDictionary new.
            self reinitializePool]

reinitializePool
    PoolUserLanguageBinding := PoolDictionary new.
    self reinitializePoolWithMySettings

reinitializePoolWithMySettings
    PoolUserLanguageBinding at: #hello put: 'Hallo'.
    PoolUserLanguageBinding at: #goodbye put: 'Bye'.
    PoolUserLanguageBinding at: #tagEnd put: ':'.
    PoolUserLanguageBinding at: #quoteStart put: ''''.
    PoolUserLanguageBinding at: #quoteEnd put: (String with: $')
```

För att testa det hela skriver vi också en instansmetod där vi som framgår kan referera till nycklarna i PoolUserLanguageBinding direkt.

test pool



Om vi testar det hela så får vi följande resultat:

```
PoolUserEnglish new quotedMessage: 'pool test'
```

⇒*'Hallo*
if we quote pool test we end up in: 'pool test'

Bye'

Nu definierar vi en ny klass som också använder poolvariabeln PoolUserLanguageBinding.

```
Object subclass: #PoolUserSwedish
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: 'PoolUserLanguageBinding '
    category: 'Test-Pool'
```

För att enkelt kunna byta till svenska värden för poolvariabeln konstruerar vi också en klassmetod i vilken vi gör de ”svenska” bindningarna.

```
initialize-pool

reinitializePoolWithMySettings
    PoolUserLanguageBinding at: #hello put: 'Hallå'.
    PoolUserLanguageBinding at: #goodbye put: 'Adjö'.
    PoolUserLanguageBinding at: #quoteStart put: (String with: (Character value:
    16rAB)).
    PoolUserLanguageBinding at: #quoteEnd put:  (String with: (Character value:
    16rBB))
```

Vi konstruerar en instansmetod quotedMessage: i den här klassen också, men vi har bytt ut de engelska ledtexterna mot svenska.

```
test pool

quotedMessage: aMessage
    ^hello,'
    om vi skriver ', aMessage, ' inom citationstecken får vi', tagEnd, ' ', quoteStart,
    aMessage, quoteEnd, '

    ', goodbye
```

Nu kan vi pröva och se vad som sker om vi använder den här klassen istället.

```
PoolUserSwedish new quotedMessage: 'testpool'
```

⟹*'Hallo*
om vi skriver testpool inom citationstecken får vi: `testpool'

Bye'

Som framgår så används fortfarande de engelska inställningarna. Om vi vill att svenska ska gälla måste vi först definiera om värdena för de olika nycklarna så att svenska används istället. Detta kan vi göra genom att använda klassmeddelandet reinitializePoolWithMySettings

```
PoolUserSwedish reinitializePoolWithMySettings.
```

och göra om försöket

```
PoolUserSwedish new quotedMessage: 'testpool'
```

⟹*'Hallå*
om vi skriver testpool inom citationstecken får vi: «testpool»

Adjö'

vilket är vad vi önskar.

Instansvariabler för klassen

En typ av variabel som inte används så mycket, men som ändå kan vara väldigt användbar är instansvariabler för klassen. En sådan variabel är inte direkt åtkomlig för instanserna.

Har man en instansvariabel för en klass så får man en instans av variabeln för varje klass, dvs en för klassen den är definierad i och en för varje subklass.

För att definiera en sådan variabel i browsern börjar man med att trycka på knappen **class** för att indikera att metaklassen ska editeras. Om en klass är vald i klasslistan, säg klassen KlassNamn, visar sig då följande text i textfönstret:

```
KlassNamn class
    instanceVariableNames: ' '
```

För att definiera en eller flera klassinstansvariabler i klassen är det bara att som för de vanliga instansvariablerna räkna upp dem och använda menyalternativet **accept** i browsern, tex:

```
KlassNamn class
    instanceVariableNames: 'klassInstansVariabelEtt enAnnan'
```

Namnreglerna är desamma som för vanliga instansvariabler.

Exempel: Klasser för att konvertera gallon och pint till liter.

För att illustrera användningen av instansvariabel för klassen ska vi implementera klasser för att konvertera gallon och pint till liter. Ett problem är att en amerikansk gallon skiljer sig från en engelsk dito. Därför väljer vi att konstruera en klass per typ av system, dvs US respektive Imperial. Konverteringarna är så pass lika och i båda systemen gäller att en pint är en åttondels gallon så vi väljer att konstruera en abstrakt klass, SIConverter, för att beskriva gemensamma attribut och beteende. Denna uppdelning är också bra om vi vill gå vidare och understödja andra konverteringar (som inch till cm) från dessa system till SI-systemet.

Lösning: Konstruera en abstrakt klass SIConverter *som gemensam superklass till de andra två systemens klasser,* ImperialConverter *och* USConverter.

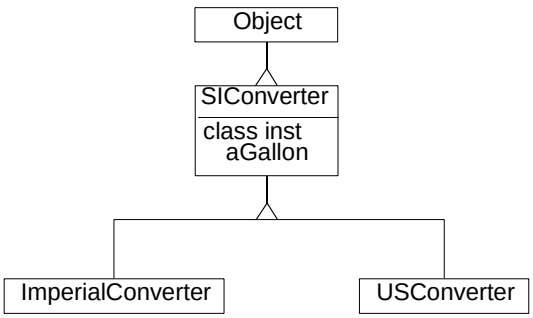
Nu är kanske den första tanken att vi borde fånga in gallon i en variabel, då detta är det enda som skiljer de två systemen åt. Vi kan inte använda en gemensam klassvariabel för detta. Men en lösning vore att konstruera en klassvariabel per konkret subklass till SIConverter. Denna klassvariabel skulle kunna ha samma namn då den inte delas mellan de två konkreta klasserna. En annan lösning vore att varje gång en instans skapas ge den värdet av gallon i en instansvariabel. En tredje lösning vore att definiera en metod per konkret klass som returnerar det aktuella gallonvärdet.

Vi känner oss inte riktigt tillfreds med någon av de föreslagna lösningarna utan bestämmer oss istället för att pröva en instansvariabel för

klassen. Därigenom kan vi definiera den på ett ställe, dvs i den abstrakta klassen, och skriva klassmetoder som tydliggör för de konkreta subklasserna att de måste initiera den med sina egna unika värden.

Om vi senare tex skulle vilja ändra precisionen för en US-gallon kan vi göra detta utan att ändra någon metod, vilket det andra av de ovan föreslagna lösningalternativen hade lett till.

Nu kan vi gå över till att definiera klasserna. Observera att de är ganska primitiva med mer eller mindre inget beteende. Men i en eventuell förlängning kan vi enkelt lägga till nya attribut i respektive klass så att varje instans skulle kunna beskriva olika storheter. Klasshierarkin för klasserna beskrivs i figur 4.7.



Figur 4.7 Klassherarki för rymdmått

```
Object subclass: #SIConverter
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'SI-Converters'
```

Vi definierar metoder för att returnera det aktuella objektets gallon eller pint uttryckt i liter. Observera hur meddelandet class utnyttjas för att anropa klassmetoder. Vi har också valt att direkt returnera en konstant från metoden pintsPerGallon då denna konstant är universell och inte kommer att förändras.

```
constants

aGallon
    ^self class aGallon

aPint
    ^self aGallon / self pintsPerGallon

pintsPerGallon
    ^8
```

På klassidan deklarerar vi aGallon som en instansvariabel för klassen.

```
SIConverter class
    instanceVariableNames: 'aGallon '
```

Vi konstruerar också några klassinitieringsmetoder, bl a för att tvinga fram en initiering av aGallon i repektive konkret subklass men också för att förbereda för en eventuell framtida utvidgning.

```
initialize

assignGallon: aNumber
    aGallon := aNumber

initialize
    self initializeConstants

initializeConstants
    self initializeGallon

initializeGallon
    self subclassResponsibility

constants

aGallon
    ^aGallon
```

Nu blir klassen ImperialConverter ganska rakt fram att skriva.

```
SIConverter subclass: #ImperialConverter
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'SI-Converters'
```

Det räcker med en klassmetod för att initiera en imperial gallon.

```
initialize

initializeGallon
    self assignGallon: 4.546
```

Det enda som skiljer klassen ImperialConverter från USConverter är värdet av en gallon.

```
SIConverter subclass: #USConverter
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'SI-Converters'
```

Det räcker att definiera om klassmetoden initializeGallon.

```
initialize

initializeGallon
    self assignGallon: 3.785
```

Nu testar vi det hela.

```
    | us |
    us := USConverter new.
    us aPint
```

⟹*0.473125*

```
    | im |
    im := ImperialConverter new.
    im aPint
```

⟹*0.56825*

4.4 Initiering av instanser

I de hittills genomgångna exemplen har initieringen av instanserna varit ett problem. Speciellt om en viss instans måste garanteras att ha instansvariabler av en viss typ eller att man i koden vill tydliggöra att vissa argument alltid måste ges vid instansieringen av klassen.

Med instansmetoder

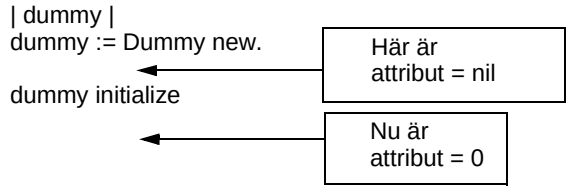
Ett sätt att åstadkomma initieringar attributen är att konstruera speciella initieringsmetoder.

En konvention är att ge en sådan initieringsmetod namnet initialize och placera den i en metodkategori med namnet initialize-release.

För att tex initiera instansvariabeln attribut till att få värdet noll, istället för skönsvärdet nil, för instanser av klassen Dummy ovan skulle vi kunna skriva följande metod:

```
initialize
    attribut := 0
```

Sedan kan vi utnyttja metoden enligt följande:



Som synes måste vi göra en uttalat initiering av objektet! Denna teknik kommer dock att förfinas i följande avsnitt.

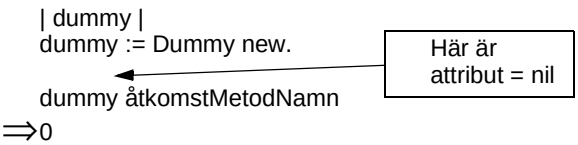
Lat initiering

En relativt vanlig teknik för att initiera ett objekts attribut är att kontrollera deras värden i inspektorerna. Då brukar man utnyttja att ett attribut alltid får värdet nil från början och om attributet vid inspektion visar sig ha värdet nil så initieras det till önskat värde innan det returneras.

Om vi tex definierar om metoden åtkomstMetodNamn kan den med denna teknik se ut som följer:

```
åtkomstMetodNamn
    attribut isNil ifTrue: [attribut := 0].
    ^attribut
```

Detta ger följande:



Resultatet blir 0 (noll) eftersom attribut binds till detta värde första gången inspektorn åtkomstMetodNamn används.

Skyddad initiering

På liknande sätt som vid lat initiering kan vi se till att en variabel endast kan ändras under vissa restriktiva omständigheter. Denna teknik kan vara användbar om man inte från början kan ge något skönsvärde men samtidigt vill ge en användare av det aktuella objektet en möjlighet att ge ett värde, men inte sedan ändra detta.

Med kontroll av om variabeln är nil

Enklast implementerar vi detta genom att utnyttja att alla variabler har värdet nil från början. Därför undersöker vi om instansvaribeln är nil och tillåter endast att ett nytt värde ges om så är fallet.

```
attribut: nyttVärde
    attribut isNil ifTrue: [attribut := nyttVärde]
```

Med eget "nilobjekt"

Ett annat problem som vi inte har löst med de föreslagna teknikerna är att om användaren faktiskt vill ge nil som värde för instansvaribalen och sedan inte tillåta att förändring sker.

Detta kan vi lösa på följande sätt:

1 Definiera en klassvariabel
2 Initiera klassvaribeln till att vara något unikt objekt
3 Initiera instansvariabeln till att peka på detta (hemliga) värde
4 Skriv om den muterande metoden

Rent praktiskt skulle detta kunna se ut något i stil med:

1 Definiera klassvariabel
  i klassdefinitionen, ge den tex namnet MyNilObject
2 Initiera den i klassmetod

```
initializeMyNilObject
    MyNilObject := Object new
```

3 Skriv instansmetod initialize som initierar instansvariabeln

```
initialize
    attribut := MyNilObject
```

4 Skriv om den muterande metoden

```
attribut: nyttVärde
    attribut == MyNilObject ifTrue: [attribut := nyttVärde]
```

Vilket fungerar bra om inga åtkomstmetoder till MyNilObject finns. Fördelen med att använda en klassvariabel framför tex att direkt göra något i stil med attribut := Object new är att vi enkelt kan skriva en metod som svarar på om variabeln är initierad eller ej genom att helt enkelt jämföra den med klassvariabeln.

Speciella klassmetoder

En annan vanlig teknik för att garantera instansvariablers värde är att konstruera speciella klassmetoder som ansvarar för att de får lämpliga startvärden.

Unära klassmetoder

Om instanser av klassen normalt inte skapas med argument, vilket skulle ha motiverat en nyckelordsmetod istället, så behöver man en unär klasskonstruktör. Denna konstruktör måste konstruera en ny instans av klassen men också anropa lämpliga initieringsmetoder på instanssidan.

Så om vi till klassen Dummy skriver en ny klassmetod zero som använder instansmetoden initialize på sidan 121.

```
zero
    ^self new initialize
```

Nu kan vi direkt konstruera nya initierade instanser av Dummy genom att helt enkelt skriva:

```
newDummy := Dummy zero
```

Klassmetod med parameter

Om man däremot måste ange ett initialt värde för ett objekt och detta värde skiljer sig från instans till instans, så bör man konstruera klassmetoder till vilka man anger dessa initiala värden. Självklart ska inte användaren av klassen behöva vara medveten om hur de givna argumenten binds till eller sparas med olika former av attribut.

Vi kan skriva följande klassmetod som tillåter oss att redan från början ge önskat värde på attribut.

```
attributValue: aValue
    | aDummy |
    aDummy := self new.
    aDummy uppdateringsMetod: aValue.
    ^aDummy
```

Nu testar vi också klassen.

```
| dummy |
dummy := Dummy attributValue: 47.
```

Nu har attribut *värdet 47*

Exempel: Konstruktör för temperaturklasserna

Nu kan vi skriva en konstruktör för de olika temperaturklasserna från sidan 111. Vi kan helt enkelt implementera följande metod för var och en av klasserna:

```
fromCelsiusValue: celsiusValue
    ^self new fromCelsiusValue: celsiusValue
```

Sedan bör vi också skriva om konverteringsmetoderna asFahrenheit och asKelvin i klassen Celsius så att vi utnyttjar den nya konstruktören.

```
asFahrenheit
    ^Fahrenheit fromCelsiusValue: self numericValue
```

Metoden asKelvin samt Fahrenheits och Kelvins asCelsius är likartade och lämnas som övningar.

Förändring av konstruktörer

När initiering av variabler fordras är det bäst att definiera den i någon konstruktör. Detta innebär att vi definierar om klassmetoderna new eller new: i den aktuella klassen. Fortfarande kan vi kombinera en sådan omdefinition med de i förra avsnittet beskrivna sätten att använda speciella initieringsmetoder.

Först skriver vi den nya konstruktören med hjälp av meddelandet basicNew, för att få en ny instans av den aktuella klassen, och instansmeddelandet initialize. Detta för att alltid få initieringar av nya instanser utförda:

```
new
    | dummy |
    dummy := self basicNew.
    dummy initialize.
    ^dummy
```

som i det här fallet också kan skrivas om genom att istället för pseudovariabeln self använda pseoudovariabeln super. Detta för att använda superklassens metod new.

```
new
    | dummy |
    dummy := super new.
    dummy initialize.
    ^dummy
```

Oftast är det senare sättet med super new att föredra då vi troligen är intresserade av superklassens eventuella kontroller och initieringar.

Ännu kortare kan vi skriva om metoden med hjälp av meddelanden i sekvens:

```
new
    ^super new initialize
```

4.5 Initiering av klasser

Nu ska vi kort diskutera hur olika klasser och deras globala variabler kan initieras. Med globala variabler menar vi klassvariabler och poolvariabler.

Initiering av klassvariabel

En klassvariabel initieras normalt i en klassmetod.

```
initializeDefault
    DefaultValue := 0
```

Om vi antar att denna metod är implementerad i klassen så kan vi initiera klassvariabeln genom att utföra följande:

```
Dummy initializeDefault
```

Om vi senare vill ändra skönsvärdet kan vi helt enkelt bara definiera om metoden och göra en ny initiering.

Initiering av klass

Om en klass har definierat ett klassmeddelande med namnet initialize och klassen skrivits ut med **file out** avslutas filen med texten AktuellKlass

initialize. När den åter läses in m h a **file in** innebär detta att meddelandet initialize också skickas till klassen. På grund av detta är det naturligt för tex klasser som behöver speciella initieringar av klassvariabler att omdefiniera denna metod. Detta så att dessa initieringar automatiskt sker om klassen flyttas mellan två imager.

Så för att beskriva hur en viss klass ska initieras, tex sätta klassvariabler till önskade skönsvärden, så skriver man en klassmetod med namnet initialize. I följande exempel visar vi hur en sådan metod kan se ut. Det enda som sker är ett anrop till två andra (egna) klassmetoder initializeDefault och initializeMenues som initierar klassvariabler för klassen samt konstruerar menyer som alla instanser ska använda.

```
initialize
    self initializeDefault.
    self initializeMenues
```

Om en omdefinition av någon av intieringsmetoderna görs bör man också initiera om klassen med hjälp av antingen metoden initialize, dvs genom KlassNamn initialize, eller om endast vissa delar är förändrade (tex menyer) genom den mer specifika initieringsmetoden (initializeMenues), dvs genom KlassNamn initializeMenues.

Det man bör tänka på är att initiering av en klass som har skrivits ut på sekundärminne är en rent textuell operation. Det vill säga att det som skrivs ut på filen avslutas med texten KlassNamn initialize. Vid **file in** så läses sedan klassbeskrivning och metoder in från filen. Sist i filen står alla klassinitieringar som behöver göras.

Exempel: Initiering av klasserna Kelvin *och* Fahrenheit.

För att initieringen av klassvariablerna i vårt tidigare temperaturexempel ska bli så smidig som möjligt skriver följande två klassmetoder i klassen Fahrenheit respektive Kelvin.

Fahrenheit klasskategori initialize

```
initialize
    ToCelsius := [:f | f - 32 * 5 / 9].
    FromCelsius := [:c | c * 9 / 5 + 32]
```

Kelvin klasskategori initialize

```
initialize
    ToCelsiusDiff :=  273.15
```

Genom att placera initieringen i metoden initialize blir det som tidigare nämnts enkelt att flytta eller sprida klassen till andra.

Initiering av poolvariabel

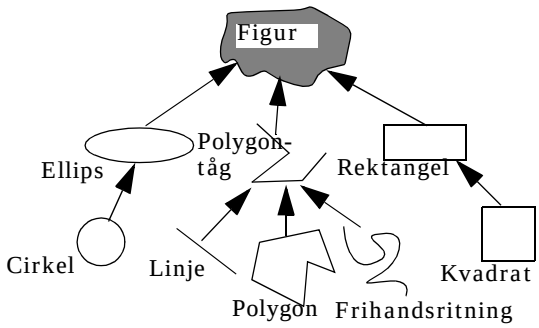
Poolvariabler är variabler som delas mellan flera olika klasser och deras instanser, även om klasserna inte ligger i samma hierarki.

Att poolvariablerna delas mellan olika klasser gör bland annat initieringen lite mer intrikat än för klassvariabler då ingen klass själv är ansvarig för detta, se sidan 113.

4.6 Arv

Nu ska vi ta en lite djupare titt på Smalltalks arvsmekanismer. Vi gör detta genom att konstruera och analysera ett antal exempel, delvis patologiska. En mer praktiskt inriktad beskrivning och användning av arv görs i avsnitt 4.7 och 4.8. Eftersom arv är fundamentalt så kommer vi dessutom tillbaka till det flera gånger i resten av boken.

Vad är då arv? För att besvara den frågan börjar vi med att förklara begreppen *är-en*, *har-en* och *del-av* (eng. *is-a*, *has-a* respektive *part-of*).



Figur 4.8 Typisk objekthierarki

En typisk *hierarki* av objekt skulle kunna se ut som i figur 4.8. Här har vi valt att placera en klass Figur högst upp i hierarkin. Denna klass är en sk *abstrakt klass* utan avsikt att instansieras. I klassen Figur placerar vi beteende och variabler som är oberoende av de konkreta subklasserna. Vi gör inte anspråk på att den valda strukturen ska vara den enda. Vi kan mycket väl tänka oss en hierarki där Polygon är superklass till Rektangel eller att dela in figurerna efter om de är slutna eller ej. Det vi vill säga med detta är att i vissa fall kan en viss hierarkisk indelning av objekten passa utmärkt men i en andra fall är en annan bättre.

Är-en

Detta begrepp utrycker att ett visst objekt är av en viss sort, tex så kan man säga att en människa är-ett däggdjur eller att en morot är-en rotfrukt som i sin tur är-en växt.

I figur 4.8 kan pilarna direkt ersättas med är-en då en cirkel är en speciell sorts ellips och ellipsen är en figur i mer generell mening. Motsvarande tolkning kan göras med de andra objekten i figuren.

Har-ett

Ett visst objekt består oftast av andra delar, vilket brukar uttryckas med att objektet har-ett visst delobjekt. Tex så har-en bil en motor och ett ansikte en mun.

Del-av

Ett objekt är också ofta del-av ett annat, tex en motor är del-av en bil eller en mun del-av ett ansikte. Denna relation är symmetrisk med har-ett relationen i den meningen att om ett objekt har-ett annat objekt så är detta objekt del-av det första objektet.

Specialisering

Specialisering genom arv innebär att en subklass är mindre generell, och mer specialiserad, än sin superklass.

I figur 4.8 är cirkeln en specialisering av ellipsen och kvadraten en specialisering av rektangeln. Linjen, polygonen och frihandsritning kan också alla ses som specialiseringar av polygontåget. Vidare, lite beroende av hur klassen Figur är implementerad, är ellipsen, polygontåget och rektangeln alla specialiseringar av figuren.

Generalisering

Generalisering är det motsatta till specialisering, dvs subklassen görs mer generell än sin superklass.

I figur 4.8 ser vi inga exempel på arv som skulle kunna sägas vara generaliseringar. Om vi tex skulle låta Cirkel och Ellips byta plats med varandra i arvsträdet skulle vi kunna säga att vi hade gjort en generalisering eftersom en ellips är mer generell än en cirkel. På samma sätt skulle vi kunna göra en Rektangel till en generalisering av en kvadrat genom att också låta dessa klasser byta plats i arvsträdet.

Aggregering

Ett alternativ till arv, men som ändå kan anses besläktat med arv, är *aggregering*. Ofta kombineras någon av de tidigare sätten att ärva med aggregering.

Om vi återigen ska relatera detta till figur 4.8 så använder troligen ett polygontåg en lista för att lagra de olika hörnen (alternativt kanterna) som beskriver den. I detta fall är denna lista utnyttjad som ett aggregat av klassen Polygontåg. Listan i sin tur använder punkter eller kantlinjer som aggregat. Om vi istället för aggregering hade försökt använda arv i Polygontåget, genom att tex subklassa en redan befintlig listklass, hade vi först varit tvungna att implementera motsvarigheten till Figur men också fått en mer statisk och mindre inkapslad struktur. Vad menar vi med detta? Jo, tänk om vi till att börja med nöjer oss med att låta en polygon bestå av en länkad lista av punkter, vilket i ”arvsfallet” skulle kunna betyda att vi gör den till subklass till standardklassen LinkedList. Om vi senare skulle komma på att en hashad lista, ett quad tree eller något annat skulle vara bättre så innebär detta ett större ingrepp i arvsstrukturen än om vi hade använt oss av ett aggregat, då vi utan att ändra den externa presentationen skulle kunne ändra listans hantering lokalt i klassen.

Multipelt arv

I en del språk, dock inte Smalltalk, finns det möjlighet att samtidigt ange flera olika superklasser. I Smalltalk har man valt att inte understödja detta vad som brukar kallas *multipelt arv*, även om det genom åren har funnits en del olika ansatser eller möjligheter till detta.

Rent konceptuellt kan användning av multipelt arv ge elegantare och kortare beskrivningar än om endast ”vanligt” arv används. En av nackdelarna med multipelt arv är dock att det inte finns någon standard utan det finns en mängd olika beskrivningar av hur detta ska gå till. En annan nackdel kan vara att det i flera situationer vid förändring av en viss klass, eller arvsträd, kan ge mer svåröverskådliga konsekvenser än om bara enkelt arv används.

Exempel på klasser som utnyttjar arvsmekanismer

Exempel: Temperaturer med abstrakt klass

Som ni har sett är stora delar av koden i de olika temperaturklasserna överlappande. Detta ska vi råda bot på nu genom att konstruera en gemensam abstrakt klass Temperature. Förutom att instansvariabeln

numericValue kommer hamna i den abstrakta klassen kommer också alla klassmetoder, förutom klassernas initieringsmetoder, och konverteringsmetoder hamna här också. De enda krav vi ställer på subklasserna Kelvin och Fahrenheit, förutom att de som tidigare ska veta sina relationer till celsiusgrader, är att de också ska veta hur ett gradtal givet i Celsius konverteras till den aktuella klassens representation och tvärt om. Vi kräver som tidigare lite mer av Celsius, nämligen att den ska kunna konvertera till de andra formaten genom att veta hur de andra formatens klasser ska anropas. I figur 4.9 beskrivs klasshierarkin för temperaturklasserna.

Klassen Temperature deklareras som subklass till Object. Vi deklarerar också att den ska ha en instansvariabel numericValue.

```
Object subclass: #Temperature
    instanceVariableNames: 'numericValue '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Temperatur'
```

Metoderna i kategorin accessing hamnar alla här. Metoden fromCelsiusValue: förlitar sig dock på att respektive subklass kan göra en konvertering från celsiusgrader till sin egen representation.

```
    accessing

    fromCelsiusValue: celsiusValue
        self numericValue: (self convertToMyRepresentation: celsiusValue)
```

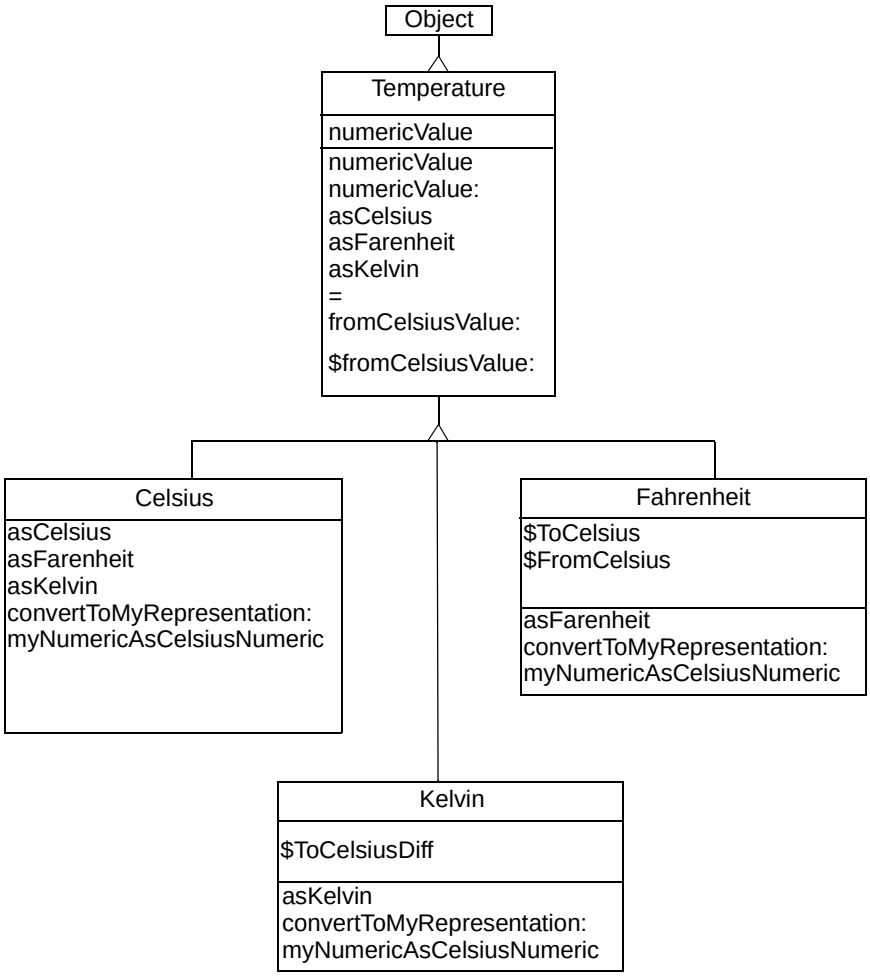
Metoderna numericValue och numericValue: är som framgår endast ”uppflyttade” till den abstrakta klassen. Annars ser de ut precis som förut.

```
    numericValue
        ^numericValue

    numericValue: newValue
        numericValue := newValue
```

Av konverteringsmetoderna är det endast asCelsius som gör något själv. De andra tar direkt hjälp av en instans av Celsius för att utföra sina respektive uppgifter.

```
    converting

    asCelsius
        ^Celsius fromCelsiusValue: self myNumericAsCelsiusNumeric

    asFahrenheit
        ^self asCelsius asFahrenheit

    asKelvin
        ^self asCelsius asKelvin
```

För att indikera att de två konverterinsgsmetoderna måste definieras om i eventuella subklasser gör vi ”stubbar” för dessa och indikerar med-



Figur 4.9 Klasshierarki för temperaturklasserna med variabler och metoder. $ markerar klassmetoder och klassvariabler.

meddelandet subclassResponsibility. Slutligen för att visa att metoderna ej är avsedda att användas utifrån så placerar vi dem i en kategori med namnet private.

```
    private

    convertToMyRepresentation: aCelsiusValue
        self subclassResponsibility

    myNumericAsCelsiusNumeric
        self subclassResponsibility
```

Nu definierar vi den enda konstruktören som behöver finnas i klassen Temperature eller någon av dess subklasser.

```
    KLASSMETODER

    instance creation

    fromCelsiusValue: celsiusValue
        ^self new fromCelsiusValue: celsiusValue
```

Nu kan vi deklarera Celsius som subklass till Temperature.

```
Temperature subclass: #Celsius
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Temperatur'
```

Vi behöver en speciell variant av alla konverteringsmetoder här, då ju alla andra klasser förlitar sig på Celsius för detta arbete.

```
    converting

    asCelsius
        ^self      Då inget behöver göras för att konvertera från celsius till celsius
                                    definierar vi av effektivitetsskäl om metoden här

    asFahrenheit
        ^Fahrenheit fromCelsiusValue: self myNumericAsCelsiusNumeric

    asKelvin
        ^Kelvin fromCelsiusValue: self myNumericAsCelsiusNumeric
```

Vi avslutar klassen med att definiera Celsius variant av de två viktiga numeriska konverteringsmetoderna. Här blir dem som synes triviala.

```
    private

    convertToMyRepresentation: aCelsiusValue
        ^aCelsiusValue

    myNumericAsCelsiusNumeric
        ^self numericValue
```

Klassdefinitionen av Fahrenheit är förutom att Temperature är superklass och instansvariabeln numericValue är ”uppflyttad” likadan som tidigare.

```
Temperature subclass: #Fahrenheit
    instanceVariableNames: ''
    classVariableNames: 'FromCelsius ToCelsius '
    poolDictionaries: ''
    category: 'Temperatur'
```

Av precis samma skäl som asCelsius definierades om för klassen Celsius definierar vi här om metoden asFahrenheit. Övriga konverteringsmetoder finns numera i Temperature.

```
converting

asFahrenheit
    ^self
```

Skillnaden mot tidigare är att de två klassvariablernas block anropas via de nödvändiga metoderna i den privata metodkategorin.

```
private

convertToMyRepresentation: aCelsiusValue
    ^FromCelsius value: aCelsiusValue

myNumericAsCelsiusNumeric
    ^ToCelsius value: self numericValue
```

Ändringarna i klassen Kelvin är helt ekvivalenta med dem som behövde göras i klassen Fahrenheit så vi kommenterar inte detta, utan listar koden direkt.

```
Temperature subclass: #Kelvin
    instanceVariableNames: ''
    classVariableNames: 'ToCelsiusDiff '
    poolDictionaries: ''
    category: 'Temperatur'

converting

asKelvin
    ^self

private

convertToMyRepresentation: aCelsiusValue
    ^aCelsiusValue + ToCelsiusDiff

myNumericAsCelsiusNumeric
    ^self numericValue - ToCelsiusDiff
```

Klassmetoderna för att initiera klassvariablerna ser fortfarande precis ut som i exemplet på sidan 126.

För att på ett elegantare sätt kunna jämföra två temperaturer, möjligen med olika temperaturskalor än då vi testade det hela på sidan 111, skriver vi också en metod = i klassen Temperature.

```
comparing

= anotherTemp
    ^self asCelsius numericValue = anotherTemp asCelsius numericValue
```

Nu kan vi testa den nya metoden.

```
| c k |
    c := Celsius new.
    c fromCelsiusValue: 10.
    k := Kelvin new.
    k fromCelsiusValue: 10.
    c = k
```

⟹*true*

Om vi inte hade skrivit om metoden så hade Object's = använts, men där är = definierad som identiskt lika med (==). Detta gjorde att inte ens två temperaturer med samma representation (tex Celsius) och gradtal (tex 47) hade uppfatts som lika innan vi skrev om den!

Metoden ~= kommer att fungera direkt då den är implementerad med hjälp av =.

4.7 Metoduppslagning

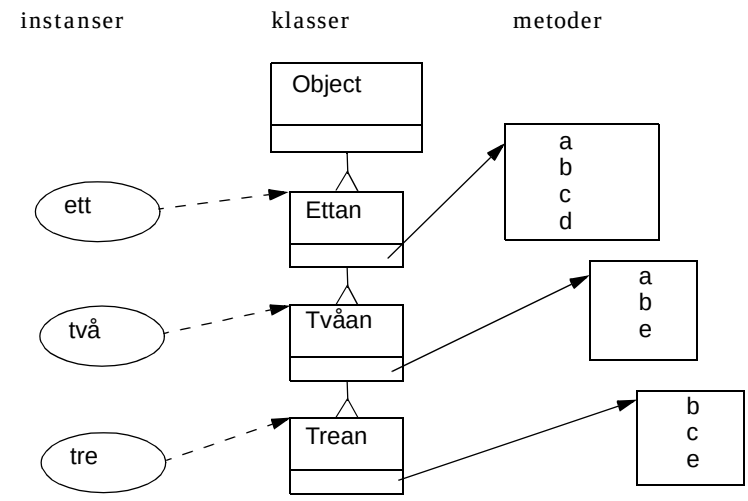
Nu ska vi titta lite på

- hur metoder är kopplade till metodnamn
- hur klasserna hanterar metodkataloger
- hur arv används i praktiken
- hur metoder i superklasser användas även om de är omdefinierade i en subklass

Metodkataloger

Till varje klass är det knutet en katalog som beskriver vilka metoder som är definierade i klassen. Vid meddelandesändning söks i första hand en metod med samma namn i den aktuella klassens metodkatalog, om ingen metod hittas där så fortsätter sökningen i dess superklass. Om ingen metod hittas där heller så forsätter sökningen vidare upp genom arvsträdet ända tills dess metoden hittas eller att vi har nått slutet av trädet, då ett felmeddelande istället aktiveras.

I figur 4.10 har vi definierat tre nya klasser vars metodkataloger också är avbildade. Låt oss anta att vi instansierar var och en av klasserna och tilldelar dem till tre olika temporära variabler enligt följande:



Figur 4.10 Arvshierarki med metodkataloger och instanser

```
ett := Ettan new.
två := Tvåan new.
tre := Trean new
```

Vi kan nu skicka olika meddelanden till objekten. Figur 4.11 beskriver vilka metoder (dvs klass>>metod) som kommer att användas om vi skickar ett antal olika meddelanden till den temporära variabeln tre (som är instans av klassen Trean).

Objekt meddelande	**Klass>>metod**	**Kommentar**
tre b	**Trean>>b**	*Metoden definierad i klassen Trean*
tre a	**Tvåan>>a**	*Metoden hittas i superklassen Tvåan*
tre d	**Ettan>>d**	*Hittas inte i Trean och inte i Tvåan men i klassen Ettan*
tre size	**Object>>size**	*Metoden hittas inte förrän i klassen Object*
tre x	**Object>>doesNotUnderstand:**	*Metoden hittas inte. Vilket resulterar i ett felavbrott.*

Figur 4.11 Meddelanden och metoder

Metoduppslagning

Som vi har sett finns det för varje klass en katalog med dess metoder. Denna katalog kan beskrivas som en uppslagsbok där varje metodnamn, eller selektor, är nyckel och motsvarande metod dess värde.

Användning av metoder i superklasser

För att anropa en metod i en superklass, även om det finns en metod med samma i den egna klassen, skriver man:

```
super metodNamn
```

Pseudovariabeln super *är ekvivalent med* self. Skillnaden ligger i (och endast i) att super indikerar att metoduppslagningen ska starta i superklassen (en nivå upp) i förhållande till den klass i vilken metoden är definierad.

```
Tvåan>>b
    ^super b
```

Om vi instansierar Trean och skickar meddelandet b till denna instans så resulterar detta i att metoden Trean>>b utförs.

```
x := Trean new.
x b => Trean>>b
```

Om vi däremot instansierar Tvåan så kommer meddelandet b medföra att först metoden Tvåan>>b utförs som i sin tur medför att Ettan>>b utförs (super b i metoden).

```
x := Tvåan new.
x b => Tvåan>>b => Ettan>>b
```

Nästa exempel illustrerar att super indikerar att metoduppslagningen ska ske en nivå upp i förhållande till den metod där variabeln används (och alltså inte har något med vilken specifik subklass som tar emot meddelandet).

Vi definierar först metoden a i klassen Tvåan. Det första metoden gör är att till motagande objekt skicka meddelandet b. Då mottagande objekt i det här fallet är representerat av super medför detta att sökningen efter metoden för b startar i klassen Ettan (oavsett om self/super är instans av Tvåan eller Ettan!)

```
Tvåan>>a
    super b.
    Transcript show: 'i Tvåan>>a'
```

Om vi nu skapar en ny instans av Trean och skickar meddelandet a till den så resulterar det först i att metoden Tvåan>>a används och ifrån denna anropas metoden Ettan>>b.

```
x := Trean new.
x a => Tvåan>>a => Ettan>>b
```

Låt oss nu definiera om de tre klasserna Ettan, Tvåan och Trean på följande sätt:

Klass: Ettan
Superklass: Object
Kategori: Hierarki

```
a
    ^self c
b
    ^self b
c
    ^'c'
d
    ^'d'
```

Klass: Tvåan
Superklass: Ettan
Kategori: Hierarki

```
a
    ^self c
b
    ^super b
e
    ^self a
```

Klass: Trean
Superklass: Tvåan
Kategori: Hierarki

```
b
    ^'b'
c
    ^super c
d
    ^super b
```

Vi sammanfattar vad som händer om vi skickar vart och ett av meddelandena till instanserna, ett, två och tre i följande figur.

Mott.	**medd.**	**väg,** (klass) metod	**resultat**
ett	a	(Ettan) a	
		(Ettan) c	'c'
	b	(Ettan) b	
		(Ettan) b	
		...	oändlig rekursion
	c	(Ettan) c	'c'
	d	(Ettan) d	'd'
	e	(Object) doesNotUnderstand:	Felavbrott, då metoden e ej hittades
två	a	(Tvåan) a	
		(Ettan) c	'c'
	b	(Tvåan) b	
		(Ettan) b	
		(Tvåan) b	
		...	oändlig rekursion
	c	(Ettan) c	'c'
	d	(Ettan) d	'd'
	e	(Tvåan) e	
		(Tvåan) a	
		(Ettan) c	'c'
tre	a	(Tvåan) a	
		(Trean) c	
		(Ettan) c	'c'
	b	(Trean) b	'b'
	c	(Trean) c	
		(Ettan) c	'c'
	d	(Trean) d	
		(Tvåan) b	
		(Ettan) b	
		(Trean) b	'b'
	e	(Tvåan) e	
		(Tvåan) a	
		(Trean) c	
		(Ettan) c	'c'

Figur 4.12 Metoduppslagning, exempel

Vad händer om en metod med visst namn inte hittas?

Som vi nämnt tidigare sker ett felavbrott om inte ett meddelande med aktuellt namn hittas i det mottagande objektets klass eller någon av dess superklassers metodkataloger. Följande frågor kan man då naturligt ställa sig:

a) Vad innebär det att ett felavbrott sker, dvs vad händer?
b) Kan vi på ett smidigt sätt hantera sådana situationer. Kan vi tex beskriva vad som ska hända om ett meddelande inte förstås?
c) Om felet indirekt beror av en av våra instanser, tex genom att ett felaktigt argument har givits till en metod, kan vi då snyggt ta hand om felet?

Svaret på dessa frågor är att:

a) Om ett meddelande ej förstås av ett visst objekt så resulterar detta i att meddelandet doesNotUnderstand: skickas till objektet. Argumentet till felmeddelandet är en instans ur klassen Message, som innehåller metodnamn och parametrar till metoden. Det objekt som är mottagare av meddelandet som resulterade i felet får vi som vanligt tag i via pseudovariabeln self.
b) Vi kan helt enkelt definiera om metoden i klasser där vi vill ha ett annat beteende. Mer detaljer i kapitel 6.
c) Vi kan ta hand om sådana fel också genom att använda Smalltalks undantagsmekanismer (eng. exception). Detta beskrivs i kapitel 12.

4.8 Abstrakta klasser

En *abstrakt klass* är en klass som man inte skapar instanser av men dess subklasser kan ha instanser. En *konkret klass* kan däremot instansieras. Med *kan* menar vi att Smalltalk inte direkt hindrar en instansiering, utan i så fall måste vi omdefiniera reaktionen på konstruktorerna i den aktuella klassen.

Ett typexempel på en abstrakt klass har vi redan sett i det tidigare temperaturexemplet på sidan 129. Där konstruerade vi en abstrakt klass Temperature som bestod av gemensam datastruktur och beteende för sina två konkreta subklasser.

Ofta konstrueras en abstrakt klass för att implementera beteende eller strukturer som är gemensamt i flera olika klasser i ett arvsträd. Ibland konstrueras en abstrakt klass för att beskriva de konkreta klassernas gränssnitt. En abstrakt klass kan också användas för att dölja plattformsspecifika skillnader, där man alltid instansierar genom meddelan-

den till den abstrakta klassen men instanserna kommer ändå tillhöra konkreta subklasser. Det senare används tex av den abstrakta klassen Filename som beroende av plattform och omgivning använder olika konkreta subklasser.

Speciella tekniker i den abstrakta klassen

I Smalltalk finns det vissa meddelanden som är avsedda att användas i en abstrakt klass för att indikera att aktuell metod faller inom ramen för den konkreta klassens ansvarsområde. Det finns också metoder som används för att ange att en viss metod inte är tillämpbar i aktuell klass eller dess subklasser.

För att i en abstrakt klass kunna beskriva ett gemensamt gränssnitt men ändå speciellt indikera att vissa metoder ligger inom de konkreta subklasssernas ansvarsområde finns, det ett meddelande subclassResponsibility. Som tex i klassen Temperature.

```
convertToMyRepresentation: aCelsiusValue
    self subclassResponsibility
```

I andra situationer kan det tänkas att en klass inte kan använda vissa av de ärvda metoderna. I en sådan situation kan man markera dem med meddelandet shouldNotImplement. Metoden add: i klassen ArrayedCollection är definierad på detta sätt.

```
add: newObject
    self shouldNotImplement
```

Metoden ärvs från klasen Collection, som i sin tur har använt subclassResponsibility, för att indikera att subklasser måste definiera om metoden i fråga. Men add: har inte någon mening för ArrayedCollection eller dess subklasser då de är av typen statiska vektorer.

Om möjligt bör dock denna form av arv, ofta baserad på återvinning av kod och inte av semantik, undvikas då det kan göra programmen mer otydliga och svårbegripliga. Fast det finns situationer där det istället kan förtydliga, tex vid omdefinition av new för att tvinga fram att annan konstruktör används.

Konstruktörer i abstrakt klass

Trots att en abstrakt klass inte ska ha några instanser kan konstruktörer med fördel definieras i den abstrakta klassen.

Exempel: Den abstrakta klassen Temperature *förhindrar instansiering av sig själv*

Vi vill både ha och äta kakan så den abstrakta klassen Temperatur ska definiera alla konstruktörer men inte tillåta några instanser av sig själv. Till skillnad från verklighetens kakätande går detta att åstadkomma i Smalltalk. Vi implementerar helt enkelt om konstruktören new i klassen Temperature. Metoden ska kontrollera om det mottagande objektet är klassen Temperature och i så fall generera ett felavbrott. I annat fall konstrueras som vanligt en instans av mottagaren (dvs någon av de konkreta subklasserna) genom att använda närmaste new-konstruktör högre upp i hierarkin.

```
new
    self = Temperature ifTrue: [self error: self class name , ' är en abstrakt klass!'].
    ^super new
```

4.9 Kommentarer, råd och tips

Nu ger vi några kommentarer med vissa tips och råd. Men då detta inte är en bok i analys, design och modellering så ger vi endast korta och ganska generella råd. Den intresserade läsaren hänvisas till den relativt rika litteraturen inom detta område, se referenser sidan 455.

Modellera först, koda sedan

Objektorienterad programmering inbjuder till ett starkt strukturellt tänkande. Då allt i Smalltalk är objekt som tillhör klasser så kan man kanske ibland få intrycket av att all struktur kommer automatiskt. Men så är det inte utan alla problem fordrar att man tänker igenom dem ordentligt. Inte minst behöver man hitta vad som är objekt och klasser i modellen, vad som redan finns att tillgå, vilka metoder som behövs, osv.

Det generella råd vi kan ge är att man innan man börjar koda åtminstone bör försöka göra några skisser över de objekt, klasser och relationer som man anser ingå i problemet. Sedan bör man fundera över vilka metoder som hör till vilka klasser och kanske skriva de viktigaste i skissen. Vidare bör man ”torrköra” några typiska problem eller sekvenser redan på papperstadiet och fundera på vilka objekt, klasser och metoder som är inblandade i detta. Ofta leder detta analys och designarbete till att man upptäcker att vissa klasser, relationer och metoder bör designas om. Man upptäcker att vissa klasser eller metoder inte alls ska vara med, att nya ska läggas till, eller att arvshierarkierna borde konstrueras om.

I vissa skeden kan det vara fruktbart att konstruera prototyper, speciellt om man inte känner sig riktigt familjär med problemet eller om kraven inte är entydiga. Ett annat dilemma är att man kanske inte är riktigt säker på tekniken (läs Smalltalk) eller vad klassbiblioteket erbjuder. I dessa situationer kan man med fördel utveckla vissa prototyper fast man bör ändå försöka att analysera och designa problemet så fristående från dessa som möjligt.

Sträva efter specialisering

I de flesta situationer är det konceptuellt bättre att sträva efter specialicering istället för generalisering. Denna strävan kan dock komma i konflikt med klassbibliotek och gammal kod, där stora delar av koden redan är skriven och uttestad.

Tumregeln är mest tillämpbar i en analyssituation där man inte utgår från några existerande objekt eller i helt nya system där det inte redan finns någon kod eller design att utgå ifrån.

När använder vi arv och när använder vi aggregering

Aggregering är i många sammanhang bättre än arv. En orsak till detta är att gränssnittet mot objekten blir tydligare samt att beteendet blir mer inkapslat än vid arv. I andra sammanhang är arv mellan olika typer av objekt väldigt naturligt, då de tex beskriver variationer på samma tema, och bör då också utnyttjas.

Att använda arv gör det ofta enkelt och snabbt att utveckla kod, speciellt prototyper.

Sök gemensamt beteende och struktur

Ur klasser som beter sig likartat eller har överenstämmande struktur kan ofta abstrakt beteende på en högre konceptuell nivå hittas. I dessa fall kan vi ofta med fördel konstruera (en eller flera) gemensamma abstrakta klasser, som då innehåller överlappande delar. Subklasser till en abstrakt klass konkreteiserar sedan de respektive klassernas speciella beteende eller struktur.

Utveckla prototyper

Om ett program eller system kan utvecklas från existerande komponenter kan utvecklingstid och kostnader avsevärt reduceras. Genom att

använda ett objektorienterat språk och arvsmekanismer kan man med fördel snabbt utforska och testa nya ideer och algoritmer.

Då alla Smalltalksystem, speciellt VisualWorks, har ett vädigt rikt klassbibliotek kan man ofta hitta en klass som så när uppfyller det man önskar. Genom att konstruera subklasser till denna komponent kan man enkelt testa alternativa lösningar. Senare med en mer fullständig prototyp kan man på ett elegant och smidigt sätt konstruera alternativa förslag genom att återigen använda arv och subklassning. Om man gör förändringarna i en ny klass istället för i ”originalet” får man också en tydlig, och isolerad, dokumentation av vad som verkligen skiljer från den tidigare lösningen.

Vi vill till sist bara ge ett varningens ord relaterat till prototyputveckling. Risken är stor att man vid konstruktion av prototyper efter ett tag får program där man inte längre har kontroll över strukturen. Det kan ändå vara ganska lockande att behålla prototypen, som man kanske har spenderat lång tid och mycket möda på, och utveckla produkten direkt från den. Ofta biter man dock sig själv i svansen om man försöker göra detta! Istället bör man ta med sig erfarenheten av detta utvecklingsarbete och göra en ny analys och design av problemet. Smalltalks kraftfulla mekansimer för att kopiera, flytta och modifiera kod gör det enkelt att på ett strukturerat sätt återanvända stora delar av prototypen. I slutänden visar det sig ofta att det var mödan värt. Programmet har fått en bättre struktur och tid har sparats.

Utveckling botten upp eller stegvis förfining?

Vanligen görs designen av ett system mha *stegvis förfining*, men mjukvara skrivs ofta enligt en *botten upp modell*. Detta kan vara ett dilemma. Men genom att använda arv så löser sig många av dessa problem. Man kan enkelt konstruera klasser på en högre nivå i arvsträdet som beskriver en mer övergripande struktur men samtidigt arbeta med mer konkreta klasser längre ned i trädet för att stegvis beskriva och utveckla detaljer.

Fördelar med arv

Nu har vi pratat om arvsmekanismer så det kan vara lämpligt att kristalisera ut fördelarna med denna teknik.

Återanvändbarhet

Vid arv återanvänds den kod som definieras av superklassen. Superklassens beskrivning och kod, särskilt om den kommer från ett välkänt klassbibliotek, är säkerligen också väl uttestad. Därigenom får subklasserna direkt tillgång till fungerande och tillförlitliga algoritmer och beteenden. Detta gör att man vid konstruktion av en subklass i huvudsak kan koncentrera sig på dess speciella beteende.

Andra fördelar är att koden blir mer enhetlig och lättläst av andra, som ju troligen också använder samma basmängd av klasser från klassbiblioteket.

Underhåll och förändring

Av liknande skäl som vid återanvändbarhet kan underhållskostnaderna för färdiga produkter reduceras genom att koden i de flesta program baseras på kända och uniforma klassbibliotek.

Som vi berörde då vi diskuterade prototyper är ofta system baserade på arv enkla att på ett strukturerat sätt förändra eller finjustera. Detta genom systemets möjligheter till att konstruera nya subklasser där endast nytt eller förändrat beteende behöver beskrivas i förhållande till de superklasser de är baserade på.

Att dela kod

Kod kan med fördel delas mellan olika programmerare genom att moduler används. I Smalltalk kan detta göras genom att programmerarna använder gemensamma klasser. En skillnad mot det rena modulbegreppet är dock att det finns möjligheter att enkelt förändra beteendet av den delade koden utan att förändra den! För att åstadkomma detta kan återigen arv utnyttjas, där en ny klass konstrueras som subklass till någon av de delade klasserna. I den nya klassen kan sedan specialiceringar implementeras. Om man upptäcker att något måste ändras i basmängden av klasser så kommer denna förändring också omedelbart påverka alla subklasser.

Tänk dig en situation där en viss basklass använder linjärsökning för att hitta vissa element. Flera klasser utnyttar den som superklass. Om någon behöver en effektivare sökalgoritm (tex hashning) så kan man istället definiera förändringen i den aktuella subklassen. Om man sedan upptäcker att alla andra subklasser till basklassen har nytta av en snabbare algoritm kan man helt enkelt flytta koden till basklassen och viola!

Självklart går det inte alltid så smidigt som beskrivet här. Det kan ju hända att förändringar i vissa subklasser är gjorda så att det inte helt isolerat går att ändra i superklassen.

Ett annat vanligt sätt att dela kod är genom att konstruera speciella klasser som innehåller gemensamt beteende för andra klasser. Ofta brukar sådana klasser vara *abstrakta klasser*, dvs klasser som ej är avsedda att instansieras. Ett typexempel i Smalltalk är klassen Collection som beskriver gemensamt beteende för sina subklasser.

Enhetliga gränssnitt

En gemensam basklass konstrueras för att beskriva subklassers yttre gränssnitt. Detta ökar både läsbarhet och sannolikheten för att de olika subklasserna blir konsekvent beskrivna.

Storlek för kod och instanser

Rätt använt kan arv användas för att både reducera kodens och instansernas storlek. Genom att gemensamt beteende för många klasser kan beskrivas i en enda basklass kan mängden kod begränsas. Jämför med det icke objektorienterade sättet där stora delar av koden ofta överlapper och alltså dupliceras. Detta beror på att det inte finns något bra sätt att återanvända den genom att abstrahera gemensamt beteende.

Även det utnyttjade utrymmet för olika instanser kan begränsas genom att på varje nivå i en hierarki endast använda tillräckligt många attribut, dvs instanserna kommer inte i onödan innehålla attribut som inte används. I icke objektorienterade språk kan detta bli besvärligare och mycket mindre naturligt än i ett objektorienterat språk.

Delvis beror möjligheterna till dessa vinster av en kombination av de objektorienterade språkens arvsmekanismer och polymorfa beskrivningstekniker, d v s möjligheten att ge olika metoder samma namn.

Kostnad och nackdelar med arv

Vi har sett att fördelarna med att ha ett system som har arv men det finns också problem.

Jo-jo-problemet

Ett välkänt problem i samband med arvshierarkier är det så kallade jojo-problemet. Detta består i att om man har en klasshierarki med metoder på flera nivåer blir det ofta så att metoder från olika klasser

används omvartannat. Detta leder till att det kan vara svårt att följa metoduppslagningen som går som en jo-jo mellan klasserna.

Prestanda

Något som ofta kommer på tal i samband med objektorienterade språk är deras prestanda. Det sägs att metoduppslagningen tar lång tid om den måste göras då programmet kör. Men med moderna tekniker för metoduppslagnign så är tiden för detta sällan något problem.

Storlek av koden

Koden kan bli onödigt stor om man endast använder en liten del av koden från klasser i superklasser. Detta är dock inget större problem i Smalltalk, då vi ändå normalt har hela klassbiblioteket definierat i omgivningen.

4.10 Exempel

Klasserna som hanter personer och adresser utvidgas med initieringsmetoder och konstruktörer

Nu har vi tillräckliga kunskaper för att förbättra klasserna Name och Person så att de tillhandahåller smidigare klassmetoder för konstruktion av dem.

Åter igen använder vi detta exempel för att repetera och fördjupa de delar av kapitlet som vi anser mest väsentliga och grundläggande. Vi utvecklar också här det hela med en stegvis förfining från att skriva nya initieringsmetoder till att konstruera fullfjädrade klassmetoder.

Klassen Name

Vi börjar med att konstruera två instansmetoder i klassen Name. Här följer vi Smalltalks konvention att benämna en metod som är avsedd för en allmänn initiering med namnet *initialize* och placera den i en kategori med namnet *initialize-release*. Metoden är avsedd att ”garantera” att de två instansvariablerna christianNames och familyName blir av lämplig typ från början och ser ut som följer:

initialize-release

```
initialize
    self christianNames: OrderedCollection new.
    self familyName: ''
```

Nu skriver vi också en metod i kategorin test som returnerar true om efternamnet är en icke-tom sträng annars returneras false.

test

```
familyNameIsValid
    ^self familyName isString and: [self familyName isEmpty not]
```

Klassen Person

Metoder i en kategori med namnet private är endast avsedda för klassens interna bruk. Då meddelandet assureName endast är avsedd för internt bruk så placerar vi den i kategorin private.

private

```
assureName
    name isNil
        ifTrue:
            [name := Name new.
            name initialize]
```

Som ni ser använde vi här *lat initiering* från avsnitt 4.4 för att initiera instansvariabeln name.

Metaklassen Name class

Nu kan vi också definiera en konstruktör för klassen Name. Vi låter mottagande objekt (här klassen Name eller någon av dess subklasser) avgöra vilken typ av instans som ska skapas genom konstruktionen self new.

```
initialized
    | tmpName |
    tmpName := self new.
    tmpName initialize.
    ^tmpName
```

Som också kan skrivas kortare med hjälp av meddelanden i sekvens.

```
initialized
    ^self new initialize
```

Fördelen med att använda self istället för Name är att eventuella nya subklasser till Name direkt kan använda konstruktören. Varför kan de inte använda den första versionen (se övning 4.1)?

Klassen Person

Metoden assureName använder lat initiering av instansvariabeln name.

```
assureName
    name isNil ifTrue: [name := Name initialized]
```

Nu lägger vi till en instansvariabel address i klassen Person. Detta kan vi helt enkelt göra genom att skriva om definitionen av klasshuvudet och använda menyalternativet **do it**, i tex ett arbetsfönster, eller **accept** i browsern.

```
Object subclass: #Person
    instanceVariableNames: 'name address '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'bok-1-person'
```

Alternativt kan vi direkt uppmana klassen att lägga till en ny instansvariabel genom att skicka meddelandet addInstVarName: med namnet på den nya variabeln som argument.

```
Person addInstVarName: 'address'
```

Klassen Address omdefinieras analogt på följande sätt:

```
Object subclass: #Address
    instanceVariableNames: 'street number city country zip phone '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'bok-1-person'
```

Klassen Address

Initieringsmetod som sätter alla instansvariabler till tomma strängar.

```
initialize
    self street: ''.
    self number: ''.
    self city: ''.
    self country: ''.
    self zip: ''.
    self phone: ''.
```

Metaklassen Address class

Ny konstruktör som utnyttjar den nya instansmetoden initialize som vi precis skrev.


```
new
    | newAddress |
    newAddress := self basicNew.
    newAddress initialize.
    ^newAddress
```

Vi kan skriva om metoden utan att använda den temporära variabeln newAddress.

```
new
    ^self basicNew initialize
```

För att underlätta för instanser av Person att använda instanser av klassen Address skriver vi följande metod, som initierar instansvariabeln address.

```
assureAddress
    address isNil ifTrue: [address := Address new]
```

För att garantera att address alltid är bunden så anropar vi alltid assureAddress.

```
address
    self assureAddress.
    ^address
```

Test

Nu kan vi testa det hela genom att utföra följande kod i ett arbetsfönster.

```
| person address |
person := Person new.
person christianName: 'Kristen'.
person surname: 'Nygaard'.
address := person address.
address city: 'Oslo'.
address country: 'Norge'.
person inspect
```

Den sista raden, person inspect, öppnar ett inspektionsfönster mot objektet person med vars hjälp man kan titta på objektets attribut. Inspektionsfönster beskrivs i detalj i avsnitt 12.2.

Metaklassen Name

Nu passar vi också på att fixa till klassen Name, dvs vi tar bort metoden initialized som inte längre behövs och gör en egen konstruktör som ser till att initieringarna utförs.

```
Name class removeSelector: #initialized          vi tar bort metoden initialized
new                                              vi skriver ny konstruktör
    ^self basicNew initialize
```

Detta är inte helt snyggt för ofta vill man redan vid konstruktionen av objektet ge dess rätta attribut därför konstruerar vi också en konstruktör till vilken vi kan ge parametrar.

```
christianNames: aCollection familyName: aString
    ^self new christianNames: aCollection familyName: aString
```

Men i den här metoden är initieringen överflödig, då initialize alltid används i metoden new. Vi väljer att ta steget fullt ut och tillåter endast att instanser av Name skapas med christianNames:familyName:.

För att åstadkomma detta skriver vi vi om new.

```
new
    self shouldNotImplement
```

Nu får vi inte glömma att ändra

```
christianNames: aCollection familyName: aString
    ^self basicNew christianNames: aCollection familyName: aString
```

Detta så att den inte längre otillåtna metoden new används utan den aldrig förändliga basicNew används istället.

Klassen Person

Nu ändrar vi också Person så att ”nya” Name används.

```
KLASSMETODER

instance-creation

new
    ^self basicNew initialize


initialize-release

initialize
    address := Address new
    name := Name christianNames: #() familyName: ''
```

OBS vi tillåter fortfarande att name är odefinierat, så att man kan skapa personobjekt utan givet namn.

Nu behöver vi inte assureName eller assureAddress längre så vi kan stryka dessa och skriva om

```
name
    ^name
```

Temperaturklasserna med klassinstansvariabel för att hantera versioner

Som ett litet exempel på en möjlig användning av instansvariabler för klassen antyder vi nedan hur dessa kan användas för en enkel form av versionshantering.

Vi tänker oss att varje klass märks med ett versionsnummer i en klassinstansvariabel version. Om vi även gör markering av versionsnummer i källkodsfilen så kan vi jämföra om filen innehåller en nyare version av koden och i så fall läsa in den.

Temperature

Då vi vill att alla temperaturklasser ska vara versionsmarkerade deklarerar vi variabeln i den abstrakta klassen Temperature.

```
Temperature class
    instanceVariableNames: 'version '
```

För enkelhets skull konstruerar vi en klassmetod för att läsa variabeln och initierar det hela i klassmetoden initialize.

```
version
    ^version

initialize
    version = nil ifTrue: [version := 1.0]
```

Fahrenheit

Som sagt nu bryr vi inte så mycket om alla detaljer utan illustrerar bara principerna, så därför nöjer vi oss med följande nya klassmetod.

```
initialize
    ToCelsius := [:f | f - 32 * 5 / 9].
    FromCelsius :=[:c | c * 9 / 5 + 32].
    version := 3
```

Kelvin

Här kontrollerar vi superklassens version (i det här fallet Temperatures) och agerar lite olika på om den är versionsmarkerad eller ej (men som sagt koden är endast illustrativ).

```
initialize
    ToCelsiusDiff := 273.15.
    self superclass version isNil
        ifTrue: [version := 0.9]
        ifFalse: [version := self superclass version + 0.1]
```

Sammanfattning

Termer

Instansvariabel i klass fungerar analogt med ”vanliga” instansvariabler, dvs lokalt per objekt (här klass).

Klassvariabel en variabel som delas mellan en klass, alla subklasser till den samt alla instanser av klassen och subklasserna.

Poolvariabel variabel som delas mellan klasser som inte (nödvändigtvis) befinner sig i samma arvshierarki.

Metaklass objekt vars instanser är klasser.

Pseudovariabel variabel vars värde inte kan påverkas av programmet. Variablerna self, super, true och nil är exempel på sådana.

Speciell variabel se pseudovariabel.

Multipelt arv arv från flera parallella superklasser, ej möjligt i Smalltalk.

Lat initiering ett attribut binds till skönsvärde först vid första anropet av dess inspektor.

Konstruktör en klassmetod som används för att instansiera klassen.

Är-en (eng. is-a) uttrycker att ett objekt är av en viss sort, tex en bil är-en farkost.

Har-ett (eng. has-a) uttrycker att ett visst objekt har ett visst delobjekt, tex en bil har-en motor.

Del-av (eng. part-of) uttrycker att ett visst objekt är del av ett annat objekt, tex en en motor är del-av en bil.

Specialisering en klass som är mer specialicerad än sin superklass, tex kan vi definiera Cirkel som subklass till Ellips.

Generalisering en klass som är mer generell än sin superklass, tex kan vi definiera Rektangel som subklass till Kvadrat.

Aggregering ett objekt består av andra objekt som används för att uträtta arbetet, dvs helheten använder delarna.

Metodkatalog en klass metoder finns i en katalog med metodnamnen som nycklar och koden som värden.

Metoduppslagning att slå upp en metod i metodkatalogen.

Abstrakt klass en klass som inte har några instanser. Används vanligen för att beskriva olika konkreta klassers gemensamma beteende och gränssnitt.

Konkret klass en klass som kan instansieras.

Jo-jo-problemet beskriver svårigheten att följa metoduppslagning om flera metoder i en klasshierarki används omvartannat.

Smalltalk

Initiering av klasser utförs vanligen i metoden initialize.

I *abstrakta klasser* markerar man att metoder ska implementeras i subklasser med meddelandet subclassResponsibility.

I *subklasser* markerar man att en metod inte ska användas med meddelandet shouldNotImplement.

Returvärdet från metod är self eller utryck efter ^ (retursymbol). Symbolen kan placeras före en metods sista sats eller innuti ett block.

Åtkomstmetoder dvs *inspektorer och mutatorer* har samma namn som attributet. Mutatorn namn är attributnamn följt av : (kolon).

Det går inte att tilldela *parametrar* objekt.

Allt är objekt!

Anrop av metod i superklass använd konstruktionen super metod-Namn. Variabeln super indikerar att metoduppslagningen skall börja en klass ovanför den klass där den aktuella metoden är definierad.

Instansvariabel för klassen klasserna är också objekt som kan ha instansvariabler.

Metodik

Modellera först, koda sedan.

Sträva efter specialisering.

Använd prototyper med arv.

Använd hellre aggregering än arv.

Övningar

4.1 På sidan 147 konstruerade vi en konstruktör initialized för klassen Name. Varför är det inte lämpligt att byta ut self mot Name? Fundera speciellt på vad som i så fall behöver göras om vi subklassar Name och hur dessa subklasser sedan kan initieras?

4.2 I klassmetoden initialize i klassen Fahrenheit på sidan 126 definierade och initierade vi en klassvariabel ToCelsius. Varför kan vi inte definiera den på följande sätt istället?

```
ToCelsius := [self - 32 * 5 / 9]
```

och sedan använda blocket genom att skicka meddelandet value till det?

4.3 Varför garanterar inte meddelandet isNil att objektet är nil?

4.4 Använd klassvariabler för att hålla reda på max- och min temperaturer.

4.5 Definiera om klassen Student från övning 2.5 som subklass till Person istället. Skriv också lämpliga konstruktörer.

4.6 Definiera återigen om klassen Student från övning 2.5 så att den nu använder Person som aggregat.

4.7 Skriv en klass Complex som hanterar komplexa tal.

4.8 Skissera en klasshierarki som hanterar fordonen personbil, lastbil, buss cykel, motorcykel, moped, segellbåt, motorbåt, sparkstötting och traktor. Tänk på ett införa abstrakta klasser där det är naturligt.

4.9 Följande uppgifter handlar om att definiera sköldpaddor med mer och mer sofistikerade egenskaper.

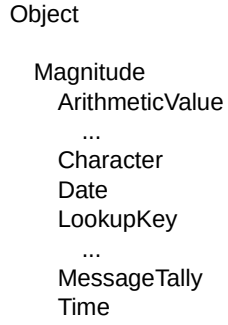
a) Gör en sköldpadda som vet var den är och i vilken riktning den är på väg och kan flytta och vrida sig.
b) Använd arv för att göra en ny förbättrad sköldpadda som vet hur långt den har vandrat (ungefär som en logg eller km-räknare).
c) Gör en riktig lyxsköldpadda som inte bara vet hur långt den vandrat utan även kan svara på hur stor area den inneslutit under sin vandring från *a* till *a* via ett antal mellanliggande punkter.
d) Gör lämpliga konstruktörer för sköldpaddorna. Tex position: aPoint och position: aPoint direction: angle.

5 Numeriska klasser

Detta kapitel besvarar bland annat följande frågor:

- Hur utför man beräkningar i Smalltalk?
- Hur hanterar man beräkningar med objekt av olika typ?
- Hur hanterar man tid?

En grundläggande strävan vid utvecklingen av Smalltalk har varit att konstruera ett system som så uniformt som möjligt hanterar alla typer av objekt. Därför är Smalltalk baserad på objekt som alla tillhör klasser vars operationer utförs genom att skicka meddelanden till dem. Speciellt tydligt blir detta om vi tex betraktar hur numeriska objekt, objekt för att hantera teckenkonstanter eller andra objekt med en naturlig ordning hanteras och definieras. I Smalltalk tillhör även dessa grundläggande typer klasser som definierar eller beskriver hur de ska hanteras.



Figur 5.1 Magnitude med subklasser

I figuren ovan illustreras arvsträdet med Magnitude och dess närmaste subklasser. De numeriska klasserna, tex de av aritmetisk natur som är subklasser till ArithmeticValue se figur 5.2; klassen som hanterar koordinater, dvs Point; klassen för att hantera tecken, dvs Character, och en del andra klasser som Time, Date och LookupKey (den senare används bland annat för att hantera associationer och hashade mängder) har det

gemensamt att de implementerar metoder som implicerar en inbördes ordning. Den generiska basen för att göra jämförelser (<, <=, > osv) ärver de alla från klassen Magnitude.

Jämförelse och tester

Gemensamt för alla de aritmetiska objekten är att de kan jämföras med meddelanden som <, <=, max:, min: och between:and:. De kan också svara på frågor om de är negativa, positiva, strängt positiva eller noll. De sista har dock ingen mening för andra subklasser till Magnitude som tex Character och Date.

Vi kan alltså jämföra objekt

```
3.5 < (1/2)
```
⇒*false*

Vi kan också undersöka om ett objekt befinner sig inom vissa gränser

```
2.5 between: 1/2 and: 4.4
```
⇒*true*

vilket också går att göra med tex teckenkonstanter.

```
$g between: $c and: $p
```
⇒*true*

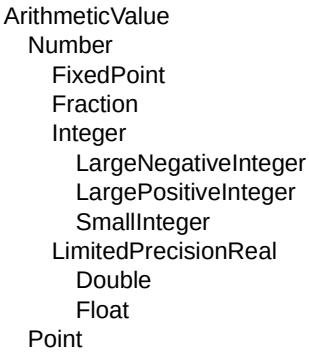
Objekten av den här typen definierar maximum respektive minimum metoder.

```
$x max: $ä
```
⇒*$ä*

5.1 Aritmetiska klasser

De aritmetiska klasserna introducerar vi genom att först beskriva några av de mest grundläggande mekanismerna och metoderna och därefter visa hur nya metoder kan skrivas.

Även de aritmetiska objekten följer Smalltalks generella prioriteritets-ordning. Det medför att parenteser behövs för att få tex multiplikation att utföras före addition på det sättet man är van vid från matematiken.



Figur 5.2 De aritmetiska klasserna

Som figuren ovan illusterar är de numeriska klasserna ordnade i en hierarki av olika klasser som beskriver objekt av likartat slag. De viktigaste av dessa klasser är klasserna Float, Integer och Fraction som är grundbyggstenar i numeriska sammanhang.

Klassen Point, som beskriver en punkt i två dimentioner, är också definierad som en aritmetisk klass. Detta medför att Point förstår metoder som är definierade i ArithmeticValue.

Aritmetik

Som namnet på klassen ArithmeticValue antyder så kan man skicka aritmetiska meddelanden till dessa objekt. De gemensamma aritmetiska metoderna är:

*, +, -, /, abs, negated och reciprocal

där de första fyra har självklar betydelse och de sista tre ger absolutbeloppet, negativa värdet respektive inversen av mottagaren.

Exempel: Aritmetik

```
4 reciprocal
```
⟹*(1/4)*

```
| x |
x := 2 negated.
x abs
```
⟹*2*

Matematiska operationer

Det finns också ett par matematiska metoder som har mening för alla aritmetiska objekt, nämligen:

raisedToInteger: och squared

som ger upphöjt till ett heltal respektive kvadraten av mottagaren.

```
4711 raisedToInteger: 17
```
⟹ *27728269858268639865783360230493363526652618962979601451519767 1*

Styrstrukturer

Det finns också metoder för att definiera slingor, nämligen:

to:, to:do:, to:by: och to:by:do:

Som exempel på detta skriver vi ut alla tal mellan en halv och tre halva med steget en tredjedel:

```
(1/2) to: 1.5 asRational by: 3 reciprocal do: [:i | Transcript print: i].
Transcript endEntry
```
⟹*(1/2) (5/6) (7/6) (3/2)*

Här ser vi också ett exempel på typkonvertering med asRational.

Tal

Det finns flera klasser som hanterar tal av olika slag. Alla dessa klasser är subklasser till klassen Number. För dessa klasser är ytterligare matematiska metoder definierade. Några av de viktigaste är:

Aritmetiska

//	heltalsdivision med avrundning mot negativa oändligheten
quo:	heltalsdivision med avrundning mot noll
rem:	resten vid division med quo: (tecken som mottagaren)
\\	modulo (resten) vid heltalsdivision

Exempel

```
-67 // 12          ⟹-6
-67 quo: 12        ⟹-5
-67 rem: 12        ⟹-7
-67 \\ 12          ⟹5
```

Matematiska

** eller raisedTo:	upphöjt till
sqrt	roten ur
squared	kvadraten

Exempel

4 ** 3.5	⇒*128.0*
7 sqrt	⇒*2.64575*
3.2 squared	⇒*10.24*

Trigonometriska

arcCos, arcSin, arcTan, cos, sin, tan, hanterar vinklar och deras inverser. Vinklarna anges i radianer.

Exempel

Float pi cos	⇒*-1.0*
3 arcTan	⇒*1.24905*

Logaritmiska

exp	exponentialfunktionen
ln, log, log:	naturliga logaritmen, $\log_{10}$ respektive $\log_n$ (n argument)

Exempel

0.3e2 exp	⇒*1.06865e13*
100 ln	⇒*4.60517*
100 log: 16	⇒*1.66096*

avrundning

ceiling, floor, rounded, roundTo:, truncated, truncateTo: (se systemet!)

2.5 ceiling	⇒*3*
2.5 floor	⇒*2*
2.5 rounded	⇒*3*
2.5 roundTo: 0.9	⇒*2.7*
2.5 truncateTo: 0.9	⇒*1.8*

Test

even	är mottagaren ett jämnt tal?
odd	är det udda?

Exempel

37 odd	⇒*true*
37 even	⇒*false*

Tecken

positive, strictlyPositive, negative och strictlyNegative, svarar på frågan om mottagaren är positiv eller negativ.

Exempel

0.0 positive	⇒*true*
0.0 strictlyPositive	⇒*false*

Konvertering

degreesToRadians	hur många radianer är ett tal givet i grader?
radiansToDegrees	hur många grader är ett visst tal givet i radianer?
asPoint	skapa punkt med både x- och y-koordinat som mottagaren
@	punkt med mottagaren som x-koordinat. y-koordinaten ges av argumentet

Exempel

45 degreesToRadians	⇒*0.785398*
Float pi radiansToDegrees	⇒*180.0*
12 asPoint	⇒*12@12*
3@4	⇒*3@4*

Heltal

Heltal i Smalltalk kan vara hur stora eller små som helst. Det finns metoder för att konvertera heltal till andra typer av tal eller tecken, beräkna största gemensamma delare, största gemensamma multipel och fakulteter. Vidare finns det en mängd metoder för bitvisa logiska operationer.

factorial	fakultet
gcd:	största gemensamma delare
lcm:	minsta gemensamma multipel
asCharacter	tecknet med mottagarens värde
timesRepeat:	upprepa argumentblocket det antal gånger som anges av mottagaren
bitOr:	bitvis logiskt *eller*
bitAnd:	bitvis logiskt *och*
bitShift:	skifta bitvis. Vänster med positivt argument, annars höger
bitInvert	komplementet
allMask:	bitmask

Figur 5.3 Några typiska metoder för heltal

Exempel

```
6000 gcd: 790                          ⇒10
6000 lcm: 790                          ⇒474000
16 bitShift: 2                         ⇒64
x := 1.
3 timesRepeat: [x := x * (x + 1)].
x                                      ⇒42
6277 bitOr: 4711                       ⇒6887
```

Den sista beräkningen kan förtydligas genom att skriva ut i basen 2.

```
6277 printStringRadix: 2               ⇒'1100010000101'
4711 printStringRadix: 2               ⇒'1001001100111'
6887 printStringRadix: 2               ⇒'1101011100111'
```

Flyttal

Flyttal finns av två olika typer nämligen enkel och dubbel precision. Tal av den första typen tilhör klassen Float och de senare klassen Double.

Flyttal i enkel precision är tal i intervallet -10^{38} och 10^{38}, med åtta eller nio signifikanta siffror. Flyttal i dubbel precision är tal i intervallet -10^{307} och 10^{307}, med 14 eller 15 signifikanta siffror.

Om inget annat anges arbetar Smalltalk med enkel precision. Om dubbel precision önskas skrivs ett suffix d efter flyttalet. Det går även att konvertera ett flyttal till dubbel precision genom meddelandet asDouble.

```
0.08d
⇒0.08d
```

```
0.08 asDouble
⇒0.0799999982118 61d
```

Som synes är det inte säkert att resultatet blir exakt detsamma om asDouble används istället för att talet direkt konstrueras i dubbel precision med suffixet d.

Bråk

Smalltalk hanterar också tal på bråkform. Klassen som hanterar dessa heter Fraction. Ett bråk består av en täljare och nämnare, båda heltal, vilket innebär att också bråk kan vara oändligt stora eller små.

Resultatet vid beräkningar med bråk förkortas alltid med hjälp av gcd.

```
14 /  18 * (12 / 5)
⇒(28/15)
401 factorial / 405 factorial
⇒(1/26507421720)
```

Klassen FixedPoint

Den sista typen av tal är de som är instanser av klassen FixedPoint. En instans av FixedPoint är ett tal med oändlig precision i heltalsdelen men begränsad precision i decimaldelen. Precisionen kan dock anges med önskat antal signifikanta siffror. Denna typ av tal används med fördel istället för flyttal om man måste garantera att en viss precision används, tex i finansiella system eller vid vissa numeriska beräkningar.

Ett tal av denna typ skapas genom att ett s skrivs som suffix till ett numeriskt värde.

```
13.456s
⇒13.456s
```

En skalfaktor, eller 10-potens, kan också anges.

```
2.4s3
⇒2400.0s
45s-6
⇒0.0000450s
```

Ett exempel där aritmetik med vanliga flyttal ger ett felaktigt resultat är följande:

```
4711.12345 / 12345 * 12345
⇒4711.12
```

Om vi däremot använder FixedPoint så blir det hela riktigt.

```
4711.12345s / 12345 * 12345
```
⟹*4711.12345s*

Ett tal kan också konvereras till FixedPoint mha meddelandet asFixedPoint:, där argumentet anger precisionen.

```
0.1 asFixedPoint: 10
```
⟹*0.1000000000s*

Så om vi rekapitulerar konverteringen av ett flyttal från enkel till dubbel precision, där vi ”förlorade” information, så kan vi istället gå omvägen via FixedPoint, och därmed undvika avrundningsfel:

```
(0.08 asFixedPoint: 1) asDouble
```
⟹*0.08d*

Utvidgning

I det här avsnittet ska vi beskriva hur man enkelt kan utöka de befintliga aritmetiska klasserna med nya metoder. Vi vill dock påminna om att man bör vara lite försiktig vid utökning av systemklasserna och tänka sig för ordentligt innan man gör det.

Följande formel definierar a över b:

$$\binom{a}{b} = \frac{a!}{(a-b)! \cdot b!}, \text{ där } a, b \in N \text{ och } a \geq b$$

Definitionen kan enkelt implementeras i Smalltalks klass Integer, enligt följande:

```
over: b
    "returnerar self över b"
    ^self factorial / ((self - b) factorial *  b factorial)
```

För enkelhets skull har vi inte gjort någon kontroll[1] av om de ingående talen är heltal större än noll eller om mottagaren (self) är större än argumentet b.

Nu kan vi direkt använda metoden med ett heltal som mottagare och ett annat heltal som argument. Om vi tex vill räkna ut hur många olika

[1] Sådana tester görs bäst i gränssnittet mot användaren. Gränssnittet måste ha kännedom om lämpliga sätt att interaktivt hantera fel. I kapitel 12 visar vi på hur fel kan hanteras och signaleras mellan programmens olika delar.

lottorader det finns, dvs på hur många sätt vi kan välja 7 siffror av 35, så kan vi göra det:

```
35 over: 7
```

⇒*6724520.*

Som ytterligare ett exempel på en metod som med fördel implementeras i klassen Integer konstruerar vi fibonacci vars avsikt är att returnera det n:te Fibonacci-talet som definieras med:

$$f_0 = f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

```
fibonacci
    self <= 1 ifTrue: [^1].
    ^(self - 1) fibonacci + (self - 2) fibonacci
```

Som framgår har vi använt oss av en rekursiv definition. I övning 4.2 ombeds läsaren att skriva om metoden så att den blir effektivare. Nu prövar vi det hela:

```
30 fibonacci
```

⇒*1346269.*

Observera att redan med detta lilla fibonacci-tal tar det, pga den explosivt rekursiva algoritmen, relativt lång tid att göra beräkningen!

5.2 Character

VisualWorks använder en utvidgad mängd av tecken med teckenkoder från 0 (noll) tom 65535. För tecken mellan 128 och 65535 används Xerox Character Code Standard (XNSS 058710), förutom att koderna för dollar- ($) respektive valutatecken (¤) överenstämmer istället med ASCII. Tecken med koder mellan 128 och 255 överensstämmer med ISO 6937 med samma reservation för ($) och (¤).

När ett tecken konstrueras som en separat entitet så används antingen dess litterala form, metoden value: eller konvertering från heltal.

Tecken	Litteral form	value:	Från heltal
a	$a	Character value: 97	97 asCharacter
6	$6	Character value: 54	54 asCharacter
#	$#	Character value: 35	35 asCharacter
$	$$	Character value: 36	36 asCharacter

Figur 5.4 Teckenkonstruktion

Grundläggande operationer

Några av de vanligaste metoderna är:

jämförelse

< och = plus de från Magnitude ärvda >, <=, max:, between:and: mfl.

konvertering

asInteger, som ger tecknets teckenkod
asLowercase, ger gemena (liten bokstav) motsvarigheten
asUppercase, ger versala (stor bokstav) motsvarigheten

test

isAlphabetic, tillhör bokstaven det engelska alfabetet?
isAlphaNumeric, alfabetisk eller siffra?
isDigit, är det en siffra?
isLetter, alfabetisk inklusive nationella specialiteter?
isLowercase, är det en gemen?
isSeparator, är det ett separatortecken, dvs cr, tab eller dyl?
isUppercase, är det en versal?
isVowel, är det en vokal?

Konstanter

Förutom en mängd olika meddelanden för att skapa olika tecken definierar klassen också en hel del teckenkonstanter som cr, tab, backspace, space, esc mfl. Dessa konstanter erbjuds som klassmetoder.

5.3 Time

Instanser av klassen Time representerar en specifik tid på dygnet. Upplösningen hos en instans av Time är normalt sekunder. Klassen Time tillhandahåller också metoder för att läsa av systemklockan och mäta exekveringstiden för objekt (vanligen block) i millisekunder.

Vi konstruerar några typexempel som utnyttjar några av Times grundläggande metoder. För mer detaljer hänvisas läsaren till systemet.

En instans av Time kan bland annat konstrueras från en angivelse i sekunder från midnatt.

```
Time fromSeconds: 247
```

Med hjälp av Time kan man på ta reda på vad klockan är.

```
Time now
```

Det går också att ange tider på många olika sätt genom att använda strängar. Konstruera en instans av Time som representerar klockan 17:25:30. Klassmetoden readFrom: tar som argument en ström över en sträng som innehåller det aktuella klockslaget. En ström över en sträng kan skapas på följande sätt: ReadStream on: aString

Lösning:

```
Time readFrom: (ReadStream on: '17:25:30')
```

Alternativt med engelsk/amerikansk tidsangivelse

```
Time readFrom: (ReadStream on: '5:25:30 pm')
```

Tider är subklasser till klassen Magnitude och kan därmed jämföras. Nya instanser av klassen kan skapas genom operationer på andra instanser av klassen.

Använd Time för att ta reda på hur många timmar, minuter och sekunder det är mellan klockan 8:10:30 på morgonen och 17:00 på eftermiddagen .

```
(Time readFrom: (ReadStream on: '17'))
    subtractTime: (Time readFrom: (ReadStream on: '8:10:30') )
```

⇒*8:49:30 am*

För att hantera tider och datum på kanonisk form kan de direkt konverteras till sekunder. Hur många sekunder är det mellan de två tidpunkterna i exemplet ovan?

```
((Time readFrom: (ReadStream on: '17')) subtractTime: (Time readFrom:
(ReadStream on: '8:10:30') )) asSeconds
```

⇒*31770*

Smalltalk har också en egen klocka med en upplösning på en tusendels sekund. Hur lång tid har gått sedan Smalltalks tusendelssekundklocka startade eller slog över och blev noll igen?

```
Time millisecondClockValue
```

Det finns också en tid då allting började! Hur många sekunder har gått sedan början av 1901?

```
Time totalSeconds
```

Man kan använda klassen Time för att mäta hur lång tid det tar att exekvera ett block. Hur lång tid tar det att exekvera 300!/200!

```
Time millisecondsToRun: [300 factorial / 200 factorial]
```

⇒*54*

(på i det här fallet en SUN SPARC station 10)

För enkelhets skull kan vi får reda på både dagens datum och tiden med ett enda meddelande.

Vad är det för datum och vad är klockan just nu?

```
Time dateAndTimeNow
```

⇒ *#(7 June 1995 12:38:27 pm)*

5.4 Date

Instanser av klassen Date representerar en viss dag sedan starten av den Julianska kalendern. Klassen Date kan användas till allt möjligt som har med datum att göra; exempelvis ge oss dagens datum, säga oss om det var skottår, eller ge oss veckodag för ett visst datum (detta bla genom att utnyttja klassen Time's möjlighet till att svara på hur många sekunder som gått sedan 1 januari 1901).

Vi illustrerar hur klassen Date kan användas för att lösa vissa uppgifter som har med datum att göra. Konstruera en instans av klassen Date som representerar datumet 1 september 1939

```
date := Date newDay: 1 month: #September year: 1939
```

Alternativt kan en instans av Date också konstrueras från en Stream på följande format:

Format	**exempel**
<dag> <månadsnamn> <år>	1 Sept 1939 1-SEP-39
<månadsnamn> <dag> <år>	Sept 1, 1939
<månadsnummer> <dag> <år>	9/1/39

Figur 5.5 Konstruktion av instanser av Date

dvs vi kan också konstruera instansen date enligt följande:

```
date := Date readFrom: '1 Sept 1939' readStream.
date := Date readFrom: '9/1/39' readStream
```

Här har vi utnyttjat det förkortade skrivsättet:

```
aString readStream
```

istället för att skriva:

```
ReadStream on: aString.
```

Vilken veckodag var det den veckodag som representeras av variabeln date ovan?

```
date weekday
⇒#Friday
```

Vilket ordningsnummer har den månad som representeras av date?

```
date monthIndex
⇒9
```

Vilken dag i ordningen det aktuella året var det?

```
date day
⇒244
```

Hur många dagar var det kvar på året och hur många dagar var det totalt detta år?

```
date daysLeftInYear
⇒121
date daysInYear
⇒365
```

Var det skottår det datumet som representeras av date?

```
date leap ~= 0
```

Exempel: Datum

a)Vad är det för datum idag?
b)Hur många dagar har gått mellan date och today?
c)Hur många sekunder har gått mellan date och today?

a) today := Date today.	*med tex 21 September 1994 som svar*
b) today subtractDate: date	*med svaret 20109*
c) today asSeconds - date asSeconds	*med svaret 1737417600*

Om man tex vill veta vilket ordningnummer en viss veckodag har, antal dagar i en viss månad ett visst år och liknande saker finns det klassmetoder som går att använda, dvs man behöver inte instansiera klassen först. Det finns inte något stöd att läsa in ett datum på formen <år><månad><dag>, däremot finns det sätt att styra hur ett datum presenterar sig som textsträng. Man anger då med en vektor hur datumet ska presentera sig där

Position	Beskrivning
1-3	1 ger dag, 2 månad och 3 år
4	i denna position skrivs det tecken som ska separera dag, månad och år
5	här anges formatet för månad. Där anger 1 med ordningsnummer, 2 första 3 tecknen och 3 hela namnet
6	formatet för året där 1 ordnigsnummer, 2 ordningsnummer \\ 10 (dvs resten vid division med 10)

Figur 5.6 Datumformat

Exempelvis så betyder #(3 2 1 45 1 2) att (de sista två siffrorna av) året ska skrivas först, därefter månad med sifferkod och slutligen dagen (alltid med siffror!). De olika delarna ska vara åtskilda med bindestreck (teckenkod 45).

Dvs

```
today printFormat: #(3 2 1 45 1 2) ger '94-9-21'
```

Om man däremot vill skriva today som 940921 är det enklast att skriva en egen rutin.

```
yearString := (today year \\ 100) printString.
monthString := today monthIndex printString.
today monthIndex < 10 ifTrue: [monthString := '0', monthString].
dayString := today dayOfMonth printString.
today dayOfMonth < 10 ifTrue: [dayString := '0', dayString].
yearString, monthString, dayString
```

⇒*940921'*

5.5 Aritmetik med blandade typer

Alla operationer på aritmetiska objekt är beskrivna av klasser i Smalltalk. Ofta vill man utföra operationer med aritmetiska objekt av olika typ, som tex:

```
2.3 + 5 * (3/2)
```

där vi först adderar ett heltal till ett flyttal och sedan multiplicerar resultatet av detta med ett bråk förväntar vi oss att få ett slultligt resultat i form av ett flyttal.

I vissa språk finns givna typkonverteringsregler för sådana situationer. För optimalt beteende konverteras också oftast de ingående objekten till den form som är mest generell i det aktuella uttrycket.

Hur gör då Smalltalk, där det inte finns någon möjlighet att vid kompileringstillfället avgöra ett arguments typ? Svaret är att en teknik där en linjär ordning mellan de aritmetiska objekten, typomvandling och dubbel uppslagning används. Vi ska nedan beskriva dessa tekniker i detalj och diskutera för- respektive nackdelerna med dem.

Konvertering mellan olika typer av aritmetiska objekt

Alla typer av numeriska objekt förstår en viss basuppsättning av meddelanden för att omvandla dem till andra grundläggande aritmetiska typer. De viktigaste är beskrivna i figur 5.7.

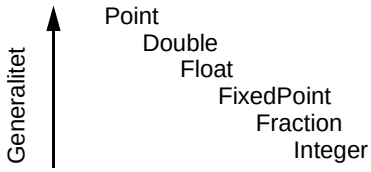
asInteger	mottagaren som heltal
asFloat	mottagaren som flyttal
asDouble	mottagaren som flyttal i dubbel precision
asRational	mottagaren som bråk
asPoint	mottagaren som punkt
@	konstruera en punkt från mottagare och argument
asCharacter	Tecknet vars värde ges av mottagaren. Fungerar bara för heltal som mottgare.
asFixedPoint:	mottagaren som FixedPoint. Antalet signifikanta siffror som argument

Figur 5.7 Metoder för att typkonvertera numeriska objekt

Generalitet

Varje aritmetiskt objekt ingår i en hierarki ordnad efter hur generellt objektet är. Denna hierarki utgår ifrån att varje typ av aritmetiskt objekt svarar med ett tal som anger dess plats i ordningen. Ett objekt som är mer generellt har ett högre ordningsnummer än ett mindre generellt objekt. Däregenom kommer ett flyttal högre upp än ett heltal, då ett heltal kan ses som ett specialfall av ett flyttal.

I figur 5.8 ser vi ett utsnitt av några av de aritmetiska klasserna ordnade på detta sätt.



Figur 5.8 De aritmetiska klasserna i hierarki efter generalitet

Om vi tex skulle vilja lägga till en klass för komplexa tal så vore dess lämpligaste plats mellan Double och Point. Detta då ett komplext tal är mer generellt än ett tal i dubbel precision men mindre generellt än en punkt (då komplexa tal ofta beskrivs som punkter i ett plan).

Dubbel uppslagning

Dubbel uppslagning (eng. double dispatching) är en generell teknik för att effektivt välja en algoritm baserad på både mottagare och argument. Det hela går ut på att ett meddelande med argument resulterar i att ett annat meddelande skickas till argumentet med den tidigare mottagaren som argument.

Typexempel

Antag att vi har tilldelat de temporära variablerna aFraction och aFloat värden på följande sätt:

```
aFraction := 10/3.
aFloat := 12.0.
```

och önskar utföra följande multiplikation:

```
aFraction * aFloat
```

där mottagaren, aFraction, är instans av Fraction och argumentet, aFloat, är instans av Float. Detta resulterar att metoden * anropas i det mottagande objektets klass, dvs i Fraction. Metoden ser ut som följer:

```
* aNumber
    ^aNumber productFromFraction: self
```

Vi ser att meddelandet productFromFraction:, med det tidigare argumentet aNumber, dvs aFloat, som mottagare och med mottagaren self, dvs aFraction som argument används. Detta meddelande är mycket mer specifikt och säger att argumentet ska vara en instans av Fraction.

I klassen som definierar metoden vet man sedan att argumentet är ett bråk och kan agera därefter. I klassen Float, som kommer att användas i det här fallet, ser metoden ut på följande sätt:

```
productFromFraction: aFraction
    ^aFraction asFloat * self
```

Det vill säga bråket omvandlas till ett flyttal och sedan används klassen Float's multiplikationsmetod. Den ser ut så här:

```
* aNumber
    <primitive: 49>
    ^aNumber productFromFloat: self
```

Primitiven utförs och svaret av primitivanropet returneras. Om vi däremot hade utfört den första multiplikationen i omvänd ordning, dvs y * x, hade först Float's metod * används. Den gör ett primitivt anrop (dvs av en metod i den virtuella makinen) som misslyckas (då argumentet är ett bråk) och därigenom utförs de efterföljande Smalltalksatserna. I det här fallet medför det att metoden productFromFloat: i klassen Fraction används, vilken ser ut som följer.

```
productFromFloat: aFloat
    ^aFloat * self asFloat
```

Ännu intressantare blir det kanske om vi utför en operation i stil med följande istället:

```
aFraction * aFraction
```

Dvs med ett bråk som både mottagare och argument. Metoden * i klassen Fraction ser ut så här.

```
* aNumber
    ^aNumber productFromFraction: self
```

Som i det här fallet leder till att följande metod i samma klass används:

```
productFromFraction: aFraction
    ^self species
        reducedNumerator: aFraction numerator * numerator
        denominator: aFraction denominator * denominator
```

Vad måste de olika klasserna förstå

Som ni sett så måste de olika klasserna vara beredda på att konvertera sina objekt, tex till flyttal, men också på att ta emot meddelanden som har med *dubbel uppslagning* att göra. Det senare gör att varje numerisk klass måste för att kunna reagera på de olika aritmetiska operationerna vara beredd på en ganska stor uppsättning meddelanden. För att inte detta ska leda till en explosion av metoder kombineras den dubbla uppslagningen med en del andra tekniker som vi beskriver nedan.

Tvingande typomvandling

Vissa metoder i systemet baseras på att de aritmetiska objekten kan tvingas att omvandlas till andra typer av aritmetiska klasser, tex att ett bråk kan omvandlas till ett flyttal i dubbel precision.

Omvandla

För att omvandla ett aritmetiskt objekt till ett annat kan coerce: användas, där argumentet ska omvandlas och bli av samma typ som mottagaren. För att göra detta brukar en dubbel uppslagningsteknik användas. Fast i dessa fall används enbart de mest grundläggande konverteringsrutinerna som asFloat och asInteger.

Som exempel så definieras coerce: på följande sätt i klassen Float:

```
coerce: aNumber
    ^aNumber asFloat
```

och på det här sätter i Integer:

```
coerce: aNumber
    ^aNumber asRational
```

Omvandling baserad på generalitet

Metoden retry:coerce: används för att beroende av vilken av mottagaren och argumentet som är mest generellt omvandla det andra till motsvarande typ. Metoden är definierad i klassen ArithmeticValue och ser ut som följer:

```
retry: aSymbol coercing: aNumber
    self generality < aNumber generality
        ifTrue: [^(aNumber coerce: self) perform: aSymbol with: aNumber].
    self generality > aNumber generality
        ifTrue: [^self perform: aSymbol with: (self coerce: aNumber)].
    self error: 'coercion attempt failed'
```

Om en viss konkret subklass till ArithmeticValue inte har konstruerat en egen metod för dubbel uppslagning så används den som är definierad i denna abstrakta superklass. Alla dess metoder för dubbelriktad uppslagning använder sedan i sin tur metoden retry:coerce: för att omvandla ett av objekten och återigen försöka utföra den aktuella beräkningen.

Ny aritmetisk klass

Om de ovan beskrivna teknikerna används är det realtivt enkelt att inkorporera nya aritmetiska klasser i systemet. Där dessa objekt kan vara mottagare till meddelanden med andra aritmetiska objekt som

argument, men också det omvända, dvs som argument till meddelanden där andra aritmetiska objekt är mottagare.

Kontruera och införliva egen klass på enklast möjliga sätt

För att så enkelt som möjligt passa in en ny aritmetisk klass definierar man dess generalitet. Sedan bör man definiera de vanliga konverteringsmetoderna dvs asInteger, asFloat mfl. Därefter definieras metoden coerce: som måste veta hur det aktuella objektet konverterar argumentet till den egna typen.

Lite mer avancerat

Ibland kan det vara bättre att även implementera alla metoder för dubbel uppslagning i den egna klassen. I vissa situationer kan det tom betyda att de existerande klassernas metoder för dubbel uppslagning ska utökas med den egna nya typen.

Eftertankar

Som vi har sett så finns det olika tekniker för att hantera samverkan mellan objekt av speciellt aritmetisk typ. Vilken är då bäst att använda? Svaret är inte entydigt utan det beror av vad som önskas och på de nya typer av objekt som ska införas. Det kan i vissa situationer vara bäst att helt och hållet använda en teknik baserad på dubbel uppslagning och i andra baserat på typkonvertering.

Den första extremen, med dubbel uppslagning, innebär att de ingående objekten inte behöver rangordnas med avseende på generalitet men till en kostnad av en metodexplosion, dvs många nya dubbel-uppslagningsmetoder måste konstrueras. Vidare måste alla klasser veta hur de kan kombineras med andra klasser.

Den andra extremen, typomvandling med coerce:, kräver i det idealiserade fallet att alla klasser befinner sig i en hierarki med en gemensam abstrakt superklass. Ibland kan det också vara svårt att avgöra ett objekts förhållande till andra objekt i generalitetshierarkin. Hur förhåller sig tex ett polynom, en determinant eller matris till ett heltal eller komplext tal?

I VisualWorks har man valt att kombinera de två olika teknikerna och därigenom ge möjligheter till att enkelt inkorporera nya typer men också optimera hanteringen.

5.6 Magnitude exempel

Slumptalsgenerator

I många sammanhang har man nytta av slumptal, eller närmare bestämt pseudoslumptal. Därför ska vi skriva en klass som ger oss möjlighet att konstruera slumptal.

Vi väljer att kalla klassen för SimpleRandomGenerator[1] den får en instansvariabel seed avsedd att hålla i ett från början givet frö från vilket nya slumptal genereras. För att från ett visst frö generera ett nytt slumptal kan man multiplicera det med en lämplig faktor och sedan ta bråkdelen av resultatet som resultat. Resultatet används sedan också som nytt frö för nya slumpdragningar. Vissa tal lämpar sig bättre än andra som faktorer vid denna process, tex vet vi att 147 är ett relativt bra tal. Men för att på bästa sätt ha möjlighet att kontrollera och förändra denna faktor, för alla slumpgeneratorer samtidigt, väljer vi att tilldela en klassvariabel det aktuella värdet.

```
Object subclass: #SimpleRandomGenerator
    instanceVariableNames: 'seed '
    classVariableNames: 'Factor '
    poolDictionaries: ''
    category: 'test-numeric'
```

Metoderna seed respektive seed: används som vanligt för att läsa respektive förändra instansvariabeln seed. Klassvariabeln Factor initieras mha av klassmetoden initialize som ser ut som följer:

```
initialize
    Den här metoden utförs alltid om vi gör fileIn på klassen, se avsnitt 11.5
    Factor := 147
```

Nu skriver vi next, den viktigaste metoden i hela klassen, vars avsikt är att ge oss ett nytt slumptal konstruerat från fröet och den givna faktorn.

```
next
    "returnerar nästa tal i slumpserien"
    self seed: (self seed * Factor) fractionPart.
    ^self seed
```

Ett problem med förfarandet att användaren måste ge ett frö är att alla slumptal som följer ett givet frö alltid är desamma, därför skriver vi också om klassmetoden new och använder Time för att generera fröet.

[1] I avsnitt 10.5 visar vi hur man kan använda den redan existerande klassen Random för att generera slumptal.

```
new
    ^self seed: ((Time millisecondClockValue + 1)  printString ,'.0') reverse
                                                                asNumber
```

Vi använder alltså systemets klocka om aktuell tid. Vi lägger till ett för att talet garanterat ska bli större än 0. Därefter omvandlar vi talet till en sträng, lägger till strängen '.0' i slutet. Nu har vi en sträng som består av ett antal siffror följt av punkt noll. Denna sträng vänder vi på och konverterar till tal. Talet blir mindre än ett och större än noll. Skälet till att invertera strängen med siffror är att talet vi får ut också kommer variera mer i de mest signifikanta siffrorna.

Vi skriver till slut klassmetoden seed:, till vilken ett frö i det öppna intervallet (0, 1) ges som argument. För att få ett mer slumpmässigt beteende bör detta frö ges med många decimaler. Metoden kan se ut som följer, där vi även har kontrollerat att fröet ligger i önskat intervall. Vi använder super new för att inte anropa det new vi definerade ovan.

```
seed: aFloat
    "returnerar första talet i en slumptalsserie om aFloat är mellan 0 och 1"
    (aFloat between: 0 and: 1)
        ifFalse: [self error: 'aFloat måste vara i det öppna intervallet (0, 1)'].
    ^super new seed: aFloat
```

Test

Nu kan vi enkelt testa klassen SimpleRandomGenerator genom att tex generera en mängd med tio slumptal

```
| bag random |
bag := Bag new.
random := SimpleRandomGenerator new.
10 timesRepeat: [bag add: random next].
bag
```

⇒*Bag (0.729019 0.0453033 0.145187 0.236252 0.342545 0.375137 0.165817 0.552628 0.35405 0.289474)*

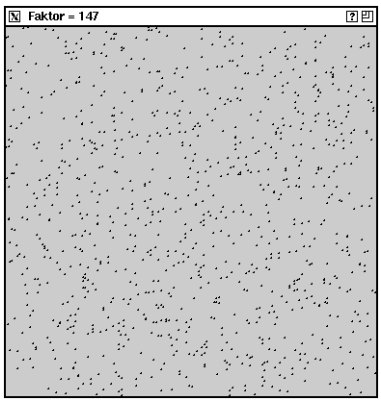
Grafiskt test

Vi gör också ett litet test där vi i dialog med en användare prövar olika värden på Factor och matar ut resultatet av en sekvens av slumptal som en serie av punkter i ett fönster på skärmen. Den här gången skriver vi testkoden som en klassmetod i klassen SimpleRandomGenerator. Att skriva en testmetod som en klassmetod är vanligt.

```
randomPoints
    "matar ut punkter slumpmässigt i ett fönster"
    "self randomPoints"
```

```
| seed window gc windSize oldFactor newFactor |
oldFactor := Factor.
window := ScheduledWindow new.
window label: 'Slumpning'.
windSize := Screen default bounds width // 3.
window openIn: (50 @ 50 extent: windSize @ windSize).
gc := window graphicsContext.
seed := 0.1234578912345d.
Factor := 147.
[(newFactor := (Dialog
        request: 'Ge faktor (avsluta med 0 eller cancel)'
        initialAnswer: Factor printString) asNumber) ~= 0]
whileTrue:
    [| rand |
    gc clear.
    Factor := newFactor.
    rand := self seed: seed.
    window label: 'Faktor = ' , newFactor printString.
    1000
        timesRepeat:
            [| randomPoint |
            randomPoint := (rand next *  windSize @
                (rand next * windSize)) truncated.
            gc displayDotOfDiameter: 2 at: randomPoint.
            gc flush]].
Factor := oldFactor.
window close
```

Ger följande prickiga resultat för faktorn 147:



Primtal

Ett primtal är som bekant ett heltal som endast är jämnt delbart med sig själv eller ett. De första fem primtalen är 2, 3, 5, 7 och 11. Primtal

spelar en viktig roll i många talteoretiska sammanhang och är avgörande för RSA-kryptering som vi tittar närmare på nedan.

Är mottagaren primtal?

I det här exemplet ska vi konstruera en metod isPrime i klassen Integer. Om isPrime skickas till ett heltal som är primtal så ska true returneras annars false. Vi kan tex utnyttja att alla primtal utom talet 2 är udda och att ett tal som ej är ett primtal måste det ha en udda delare som är mindre än heltalsdelen av roten av sig själv.

Så en möjlig lösning är:

```
isPrime
    "Returnerar true om mottagren är ett primtal"
    (self even or: [self < 2]) ifTrue: [^self = 2].
    3
        to: self sqrt truncated
        by: 2
        do: [:current | self \\ current == 0 ifTrue: [^false]].
    ^true
```

där vi har använt oss av bland annat meddelandet \\ (modulo-metoden) för att undersöka om mottagaren är jämnt delbar med argumentet, dvs med current.

Modulär exponentiering

Vi ger en metod för att beräkna x^n med modulär aritmetik, , se någon av böckerna i referenslistan på sidan 455. Vi kommer att behöva den senare i detta avsnitt då vi konstruerar ett RSA-chiffer.

Rekursiv metod för $\mathbf{x^n (mod\ m)}$

Vi utnyttjar

$$x^n = \begin{cases} \left(x^{\frac{n}{2}}\right)^2 & \text{om n jämnt} \\ x \cdot x^{n-1} & \text{annars} \end{cases}$$

och

$$a \cdot b (mod\ \mathrm{m}) = a (mod\ \mathrm{m}) \cdot b (mod\ \mathrm{m})$$

INSTANSMETODER INTEGER

```
exp: exponent mod: module
    exponent = Integer zero ifTrue: [^Integer unity].
    ^exponent even
        ifTrue: [(self exp: exponent // 2 mod: module) squared rem: module]
        ifFalse: [self * (self exp: exponent - 1 mod: module) rem: module]
```

där rem: ger resten vid heltalsdivision av mottagaren med argumentet. Exempelvis så ger 5 rem: 3 svaret 2 och 11 rem: 4 svaret 3.

Snabb metod för att hitta inverser i modulär aritmetik

Även denna metod behövs i den efterföljande versionen av RSA-kryptering. Hitta x^{-1}, dvs inversen av x i relationen: $x \cdot x^{-1} \equiv 1 (\mathrm{mod}\ m)$ där x och m kända från början.

Följande metod ger som resultat inversen av mottagaren modulo argumentet (m):

```
evaluateInverseModulo: m
    | g0 g1 u0 u1 result v0 v1 |
    g0 := m.
    g1 := self \\ m.
    u0 := 1.
    u1 := 0.
    result := v0 := 0.
    v1 := 1.
    [g1 ~= 0]
        whileTrue:
            [| tg tu y |
            y := g0 // g1.
            tg := g1.
            tu := u1.
            result := v1.
            g1 := g0 - (y * g1).
            u1 := u0 - (y * u1).
            v1 := v0 - (y * v1).
            g0 := tg.
            u0 := tu.
            v0 := result].
    result positive ifFalse: [result := result + m].
    ^result
```

Vi ger också en metod som kontrollerar att m har ett rimligt värde innan metoden ovan anropas, enligt följande:

```
inverseModulo: m
    (self > 0 and: [self < m])
    ifFalse: [^Dialog warn: 'Mottagaren måste vara i intervallet (0, m).',
            ' Nu är mottagaren = ' , self printString , ' och m = ' , m printString , '.'].
    ^self evaluateInverseModulo: m
```

RSA-kryptering

Under årens lopp har ett flertal metoder för att skicka krypterade meddelanden konstruerats. Under de senare åren har krypton som använder offentliga nycklar blivit allt mer populära. Ett berömt chiffer av denna typ är det så kallade RSA-chiffret (efter upphovsmännen Rivest, Shamir och Adelman) som bygger på en trappdörrsfunktion baserad på primtal.

Algoritm:

1 Välj två stora primtal p och q
samt tal s som är relativt primt med både (p-1) och (q-1), dvs (s gcd: p - 1) = 1 och (s gcd: q - 1) = 1.

2 Beräkna r = pq och hitta ett t sådant att

$$st = 1 \bmod (p-1)(q-1)$$

3 publicera r och s.
men hemlighåll p, q, t.

4 Kryptera med

$$E(x) == x^s \pmod r = y$$

Obs i E(x) måste x < r

5 Dekryptera med

$$D(y) == y^t \pmod r = x$$

Första försöket

Ofta är det bäst att först lösa ett problem med starkt förenklade antaganden. Senare när den förenklade lösningen är prövad och eventullt korrigerad kan man gå över på stegvis mer generella lösningar. Tack vare den objektorienterade modellen så påverkas inte användare av klassen. Vi börjar därför med en ansats utan modulär aritmetik.

```
Object subclass: #PublicKeyRSA
    instanceVariableNames: 'r s p q t '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'test-numeric'
```

INSTANSMETODER

```
p: aPrime q: anotherPrime
    p := aPrime.
    q := anotherPrime.
    r := p * q.
    self findS.
    self findT
```



```
findS
    s := (p - 1 min: q - 1) - 1.
    [s < 2 ifTrue: [s error: ''].
    (s gcd: p - 1) = 1 and: [(s gcd: q - 1) = 1]] whileFalse: [s := s - 2]
findT
    | pqProduct search i |
    pqProduct := p - 1 * (q - 1).
    search := [:n | pqProduct * n + 1 / s].
    i := 1.
    [(search value: i) isInteger]
        whileFalse: [i := i + 2].
    t := pqProduct * i + 1 / s
encryptionBlock
    ^[:x | (x raisedTo: s) \\ r]
decryptionBlock
    ^[:y | (y raisedTo: t) \\ r]
encrypt: anInteger
    "Pre: anInteger between: 0 and: r - 1"
    ^self encryptionBlock value: anInteger
decrypt: anInteger
    ^self decryptionBlock value: anInteger
```

KLASSMETODER

```
p: aPrime q: anotherPrime
    ^super new p: aPrime q: anotherPrime
```

Test

```
crypto := PublicKeyRSA p: 47 q: 79.
crypto encrypt: 67. "Ger 1302"
crypto decrypt: 1302 "Ger följdriktigt 67"
```

Förbättrad version

Prestanda vid kryptering, dekryptering och sökandet efter lämpliga värden på s och t är väldigt dålig, speciellt i en mer realistisk situtation där vi arbetar med stora primtal och värden på s och t. Lyckligtvis finns det lösningar på problemet. Vi kan utnyttja att algoritmen hela tiden använder modulär aritmetik och därmed avsevärt reducera komplexiteten av problemet.

För att återanvända beskrivningen av klassen PublicKeyRSA och betona att vår nya klass löser samma problem som denna klass så konstruerar vi en ny klass PublicKeyRSA2 som subklass till den tidigare[1].

```
PublicKeyRSA subclass: #PublicKeyRSA2
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'test-numeric'
```

Vi börjar med att definiera om krypterings- och dekrypteringsblocken så att vi utnyttjar den modulära aritmetiken vid potensberäkning. Här utnyttjar vi Integer>>exp:mod: (istället för raisedTo: och \\ som i det ”första försöket”). Detta ger följande två omdefinierade metoder:

```
encryptionBlock
    ^[:x | x exp: s mod: r]
```

och

```
decryptionBlock
    ^[:y | y exp: t mod: r]
```

För att få ett krypto som är ännu svårare att knäcka så skriver vi också om metoden findS så att vi istället söker ett s som är primtal och större än både p och q. Den nya metoden kan då skrivas som följer (där vi bland annat använder Integer>>isPrime som vi skrev tidigare, se sidan 178):

```
findS
    s := (p + 1 max: q + 1) + 1.
    [(s gcd: p - 1) = 1 and: [(s gcd: q - 1) = 1 and: [s isPrime]]]
        whileFalse: [s := s + 2]
```

Vi kan också göra en verkligt stor prestandavinst genom att definiera om findT med hjälp av Integer>>inverseModulo: från sidan 179 enligt följande:

```
findT
    t := s inverseModulo: p - 1 * (q - 1)
```

För att enkelt testa den nya klassen skriver vi om metoden new så att den genererar p och q från Time genom att använda millisecondClockValue upprepade gånger tills det returnerade värdet är ett primtal. Primtalen blir med denna strategi dock inte tillräckligt stora för att i praktiken användas vid säker kryptering. Men en förbättring av detta kan ju med hjälp av att metoden är inkapslad alltid göras senare då allt annat är uttestat i mer kontrollerbara fall.

Genom detta förfarande kan vi senare enkelt byta ut de primtal mot ännu större primtal genom att tex använda Fermats test eller Solovay-

[1] I en verklig situation hade vi nog istället ersatt den gamla klassen med den nya. Men för att ha möjlighet att jämföra de två varianterna så gör vi på detta sätt.

Strassens metod för att generera pseudoprimtal, se någon av böckerna i referenslistan på sidan 455.

```
new
    | primeOne primeTwo |
    [(primeOne := Time millisecondClockValue) isPrime] whileFalse.
    [(primeTwo := Time millisecondClockValue) ~= primeOne
                        and: [primeTwo isPrime]] whileFalse.
    ^self p: primeOne q: primeTwo
```

En klass för kryptering

Man behöver endast veta hur krypteringen ska gå till för att generera ett kryptogram för en främmande person. För att förenkla denna process tillhandahåller vi också en speciellt klass avsedd att generara sådana krypton enligt följande. Här har vi valt att speciellt understödja kryptering med hjälp av PublicKeyRSA2. Koden ser ut som följer:

```
Object subclass: #PublicKeyRSAEncrypter
    instanceVariableNames: 'r s encryptionBlock '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'test-numeric'
```

Som framgår har vi här valt att ”hålla i” krypteringsblocket med hjälp av en instansvariabel istället för som tidigare generera ett nytt block vid varje anrop av metoden encryptionBlock.

INSTANSMETODER

```
r: anInteger s: anotherInteger
    r := anInteger.
    s := anotherInteger.
    encryptionBlock := [:x | x exp: s mod: r]

encryptionBlock
    ^encryptionBlock

encrypt: anInteger
    ^self encryptionBlock value: anInteger
```

KLASSMETODER

```
r: anInteger s: anotherInteger
    ^self new r: anInteger s: anotherInteger
```

Nu är det bara att leverera klasserna, med lämpliga värden på r och s som bihang, till dem som vi vill ska kryptera meddelanden till oss.

Personer med information om ålder

För att visa hur ett felaktigt val av instansvariabel leder till problem, konstruerar vi ett felaktigt exempel och visar hur det kan åtgärdas.

```
Object subclass: #Person
    instanceVariableNames: 'name age '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'test-person'
```

Som vanligt metoder för att skriva och läsa instansvariablerna samt metoden name:age: för att samtidigt ange bådas värde.

KLASSMETODER

```
christianNames: aCollection familyName: aString age: age
    ^self new
        name: (Name christianNames: aCollection familyName: aString)
        age: age
```

Test

```
| year p1 p2 p3 p4 p5 p6 |
year := 1994.
p1 := PersonVariant christianNames: #('P1') familyName: 'F1' age: 57.
p2 := PersonVariant christianNames: #('P2') familyName: 'F1' age: 44.
p3 := PersonVariant christianNames: #('P3') familyName: 'F1' age: 25.
p4 := PersonVariant christianNames: #('P1') familyName: 'F2' age: 27.
p5 := PersonVariant christianNames: #('P2') familyName: 'F2' age: 24.
p6 := PersonVariant christianNames: #('P3') familyName: 'F2' age: 5.
```

Om nu året ändras måste vi uppdatera alla personer så att personerna blir medvetna om sin nya ålder.

```
year := year + 1.
p1 age: p1 age +1.
p2 age: p2 age +1.
p3 age: p3 age +1.
p4 age: p4 age +1.
p5 age: p5 age +1.
p6 age: p6 age +1.
p6 age
```

⇒*6*

Inte riktigt tillfredställande. Eller hur?

Förbättrat variant

Istället för att hela tiden uppdatera en persons ålder vid årsskiften använder vi ett attribut för respektive instans födelsedatum. Därmed kan vi hantera förändringar i kalendertid. Med denna förändring blir det också enkelt att hantera åldrar på ett finkornigare sätt än tidigare.

```
Object subclass: #Person
    instanceVariableNames: 'name birthday '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'test-person'
```

Som vanligt skriver vi också metoder för att skriva och läsa instansvariablerna samt metoden name:birthday: för att samtidigt ange bådas värde.

KLASSMETODER

```
christianNames: aCollection familyName: aString birthday: aDate
    ^self new
        name: (Name christianNames: aCollection familyName: aString)
        birthday: aDate
```

INSTANSMETODER

Först konstruerar vi en metod som ger en persons ålder vid ett visst datum:

```
ageAtDate: date
    (date monthIndex > self birthday monthIndex
        or: [date monthIndex = self birthday monthIndex
            and: [date dayOfMonth >= self birthday dayOfMonth]])
                ifTrue: [^date year - self birthday year].
    ^date year - 1 - self birthday year
```

Sedan skriver vi också en metod som svarar med en persons ålder vid det datum som meddelandet skickas, dvs idag.

```
age
    ^self ageAtDate: Date today
```

Test

Initieringen blir något längre men det vi tjänar på detta är värt besväret. I en mer realistisk situation har vi säkerligen ett elegant interaktivt gränssnitt som hanterar det långa skrivsättet.

Den här gången anger vi också exakta födelsedatum.

```
p1 := Person
            christianNames: #('Lars' )
            familyName: 'Svensson'
            birthday: (Date newDay: 24 month: #April year: 1937).
```

Vill veta en persons ålder vid ett visst datum är det bara att fråga.

```
p1 ageAtDate: (Date newDay: 24 month: #May year: 1995).
p1 ageAtDate: (Date newDay: 30 month: #December year: 1995).
```

Eller hur gammal den aktuella personen är just idag (sommaren 1995).

```
p1 age
```

⇒*59*

Sammanfattning

Termer

Dubbel uppslagning när ett metodanrop av generell typ specialiseras genom att skicka mottagaren som argument till argumentet.

Smalltalk

Klasser med *ordningsrelationer* är vanligen subklasser till Magnitude.

För *beräkningar* används någon av subklasserna till ArithmeticValue: Number, Fraction, Integer, Float, Double.

Date och Time används för att hantera datum och tid.

Aritmetik, tecken, datum och tid hanteras av första ordningens objekt och klasser.

Det går att skriva tal i olika *baser* och med olika *precision*. Oändligt stora och små heltal hanteras också.

Tecken hanteras normalt med Unicode.

Dubbel uppslagning används för att hantera aritmetik med objekt av olika typ.

Övningar

5.1 Vad ger 0 over: 0 som resultat om vi använder metoden på sidan 163?

5.2 På sidan 164 skrev vi en rekursiv metod fibonacci. Det är välkänt att en rekursiv metod för fibonaccitalen inte är speciellt lämpligt ur effektivitetssynpunkt. Så

a) skriv fibonnaci som en iterativ metod istället.

b) skriv om fibonacci-metoden med hjälp av det slutna uttrycket:

$$f_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}$$

5.3 På sidan 178 skrev vi en metod Integer>>isPrime m h a av ett intervall. Skriv om metoden med hjälp av whileTrue: istället.

5.4 Skriv om exp:mod: på sidan 179 med hjälp av en iterativ metod.

5.5 Hur kan vi använda meddelandet between:and: för att kontrollera om m befinner sig inom avsedda gränser i metoden inverseModulo: på sidan 179?

6 Allt är objekt

Detta kapitel besvarar bland annat följande frågor:

- Hur är Smalltalksystemets klasser uppbyggda?
- Hur är de logiska klasserna implementerade?
- Vad är ett block egentligen?
- Vad är nil egentligen?
- Hur görs klassbeskrivningar?
- Varför och vad är metaklasser?

Hittills har du lärt dig att använda många konstruktioner, utan att kanske tänka på att de kan beskrivas som meddelandesändning till objekt. I det här kapitlet ska vi titta närmare på hur Smalltalk bygger på att allt är objekt. Vi kommer också beskriva en del mer speciella klasser och konstruktioner. Detta kapitel kan i viss mån ses som ”överkurs” eftersom man även utan att förstå hur Smalltalk internt är uppbyggt kan programmera i språket, men vi tror det är av intresse att se och förstå hur man kan bygga upp ett system på en rent objektorienterad grund.

6.1 Block

En av de absolut mest använda objekten i Smalltalk är block. Vi har redan använt dem många gånger. Ofta utan att tänka på att de är instanser av en första ordningens klass. Ett block tillhör i Smalltalk-80

klassen BlockClosure. För att ta reda på vad klassen heter i ditt system kan du pröva följande[1], meddelandesekvens:

```
[] class name
```

Vi har använt block i bla slingor och villkorssatser som:

```
max := a > b
    ifTrue: [a]                    Om villkoret är sant utförs blocket med a
    ifFalse: [b]                     annars det med b, resultatet returneras
```

som jämför a och b och sedan tilldelar max det värde som är störst. I exemplet finns det två mycket enkla block [a] och [b]. Det enda som görs är att resultatet av a eller b returneras. Ett block returnerar nämligen värdet av dess sist utförda sats och ifTrue:ifFalse: returnerar i sin tur detta resultat.

Konstruktion och exekvering

Ett block skapas med hjälp av en syntaktisk konstruktion med hakparenteser

[*S*]

där *S* är en eller flera godtyckliga Smalltalksatser.

Ett exempel på ett block skapat på detta sätt är:

```
[Transcript show: 'Detta är ett block']
```

Ett blocks kropp

Ett blocks kropp följer de vanliga satsreglerna. Kroppen kan bestå av temporära variabler följt av godtyckliga satser. Detta innebär bla att ett block kan innehålla andra block och styrstrukturer, se exemplet från hailstone.

```
anInteger timesRepeat:
    [n = 1                                  blocket börjar
        ifTrue:
            [Transcript show: 'Klart'; cr.  första inre blocket börjar
            ^n]                             första inre blocket slutar
        ifFalse:
            [n odd                          andra inre blocket börjar
                ifTrue: [n := n * 3 + 1]    inre block1 i det inre blocket
                ifFalse: [n := n // 2].     inre block2 i det inre blocket
            Transcript show: n printString; cr]].
```

Exekvera block

[1] I vissa Smalltalksystem har man valt att ge denna klass ett annat namn

Ett block är en kodbeskrivning med fördröjd evaluering som utförs först när man ber om det genom att meddelandet value skickas till det.

```
[| x y |
    x := 3.
    y := x negated reciprocal.
    x / y]
```

⟹*BlockClosure [] in UndefinedObject>>unboundMethod*

```
[| x y |
    x := 3.
    y := x negated reciprocal.
    x / y] value
```

⟹*-9*

Till Smalltalks styrstrukturer ges block som argument för att beskriva vad som ska hända. Metoderna som beskriver dessa strukturer skickar vid behov value till aktuellt argumentblock. Detta återkommer vi till senare i kapitlet.

Block med en formell parameter

Ett block kan också använda en formell parameter. En sådan deklareras direkt till höger om den vänstra hakparentesen. Parametern ska föregås av ett kolon (:) och parameterlistan avslutas med ett vertikalt streck (|). Efter detta följer (precis som för block utan argument) Smalltalksatser och det hela avslutas med en höger hakparentes. Dvs det hela ser ut som följer:

```
[ : formellParameter | S ]
```

där *S* är, de möjligen tomma, Smalltalksatserna.

Ett exempel där block används som en väsentlig komponent är:

```
positive := aCollection select: [:num | num positive ]
```

Här plockas alla positiva tal från aCollection ut och bildar en ny behållare. Behållaren bildas genom att select: skapar en ny behållare, vanligen av samma typ som mottagaren av meddelandet, som består av alla objekt (num) för vilka blocket returnerar true. Slutligen tilldelas positive resultatet.

Sett lite annorlunda kan vi säga att aCollection är mottagare av meddelandet select: och argumentblocket [:num | num positive] har en formell parameter. Den formella parametern num kommer sedan i tur och ordning representera alla olika element i mängden aCollection.

Tilldela ett block till en variabel

Eftersom ett block är ett objekt så kan det tilldelas till en variabel, som tex: positive ovan,

```
f := [:x | x * x + x + 2 ]
```

eller

```
isPalindrome := [:aString | aString sameAs: aString reverse]
```

I dessa sammanhang kan det vara fruktbart att betrakta den temporära variabeln som en funktionspekare[1].

För att utföra ett block med en formell parameter skickas value: med aktuell bindning som argument.

```
y := f value: 2                    beräknar f för x lika med 2, y sätts till resultatet
```

Vi utför blocket f med parametern 2, dvs blockets formella parameter x binds till 2 och utförs.

Blocket isPalindrome anropas på analogt sätt.

```
isPalindrome value: 'Rom mor'
```

⇒*true*

Block med flera formella parametrar

Vi kan också konstruera ett block med flera formella parametrar. Dessa deklareras genom att efter den vänstra hakparantesen räkna upp dem åtskillda med kolon (:). Ett block med tre formella parametrar skulle då kunna gestalta sig som följer:

```
[ : f1 : f2 : f3 | S ]
```

där *S* är godtyckliga Smalltalksatser. Antalet formella parametrar är idag begränsat till 255 stycken.

Exempel: Flera formella parametrar

```
concatenate := [:a :b :c | a, b, c].
concatenate
    value: 'ett'
    value: 'tu'
    value: 'tre'
```

⇒*'etttutre'*

---

[1] Jämför tex med λ-kalkylen, vissa lispdialekters möjligheter att beskriva funktioner av första ordningen (tex Scheme), C's funktionspekare, ML, eller Pascal's mer restriktiva möjligheter att beskriva funktioner i argumentlistor

Returvärde från ett block

Ett block kan som vi tidigare sagt bestå av flera satser. Gemensamt för alla typer av block är att då de utförs returneras resultatet av den sist exekverade satsen.

Ett block kan, precis som en metod, tvinga fram att ett annat objekt än det som beskrivs av dess sista sats returneras genom att ett inre block returnerar ett värde. På vanligt sätt görs detta med en uppåtpil följd av det som ska returneras.

Exempel: Returvärde

```
createPalindrome := [:x | x isString ifFalse: [^'Jag vill ha en sträng'].
                          x, x reverse].
createPalindrome value: 'Ni talar bra latin -'
```

⟹*'Ni talar bra latin -- nital arb ralat iN'*

```
createPalindrome value: 4711
```

⟹*'Jag vill ha en sträng'*

Temporära variabler

Vi kan deklarera en temporär variabel inuti ett block. Sådana variabler deklareras på samma sätt som temporära variabler i metoder eller arbetsfönster, dvs inom ett par av vertikala streck i början av blockets satser, i stil med följande:

```
[ : f1 : f2 : f3 | | temp1 temp2 temp3 |
    S' ]
```

där temp1, temp2 och temp3 är tre temporära variabler och *S'* är Smalltalksatser utan inledande temporärvariabelsdeklaration.

En sådan variabel kommer, liksom en temporär variabel i en metod, vara definierad i blocket och inte synas utanför det.

Exempel: Block med flera parametrar och temporära variabler
I det här exemplet ska vi illustrera hur vi numeriskt kan beräkna derivatan av en funktion i en punkt x m h a centraldifferens

$$\lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$

där vi beskriver både funktionerna (f) och deriveringsrutinen med block.

Vi antar att den första parametern är ett block och den andra ett tal.

```
dx := [:f :x |  (f value: x + (1 / 10000)) - (f value: x - (1 / 10000)) / (1 / 20000)].
```

Funktionen, i form av ett block, ges som första argumentet och x-värdet för punkten som andra. Det är möjligt att skriva om deriveringsblocket utan att upprepa divisionen i onödan på följande sätt, där vi använder en temporär variabel (h):

```
dx := [:f :x | | h |                          deklaration av temporär variabel h
    h := 1 / 10000.
    (f value: x + h) - (f value: x - h) / (2 * h)]
```

Meddelanden för att utföra block

För att utföra ett block används ett av fem olika value-meddelanden.

Meddelande	Kommentar
value	För att evaluera block som saknar parametrar.
value:	För block med en parameter.
value:value:	För block med två parametrar.
value:value:value:	För block med tre parametrar.
valueWithArguments:	För block med ett godtyckligt antal parametrar. Argumentet till valueWithArguments ska vara en Array med rätt antal parametrar.
numArgs	Ger antalet formella parametrar i blocket.

Figur 6.1 Meddelanden till block

Antalet parametrar måste överensstämma med de formella parametrarnas antal.

Exempel: Konstruktion av rektangel från fyra heltal

```
[:x0 :y0 :x1 :y1 | x0@y0 corner: x1@y1] valueWithArguments:
                                        (Array with: 3 with: 4 with: 8 with: 7)
```

⟹*3@4 corner: 8@7*

Exempel: Deriveringsexempel med flera parametrar

En naturlig förändring av det tidigare deriveringsexemplet är att låta h vara parameter:

```
dx := [:func :x :h | (func value: x + h) - (func value: x - h) / (2 * h)]
```

Nu kan vi använda dx för att derivera funktioner, men med bättre kontroll av vad som händer.

```
result: = dx
    value: [:x | x ** 3 + x + 5]
    value: 2
    value: 1 / 1000000
```

⟹*(13000000000001/1000000000000)*

Vi beräknar derivatan för $f(x) = x^3 + x + 5$ i $x = 2$ men eftersom vi använder rationella tal så kommer resultatet att bli ett rationellt tal, efter konvertering får vi svaret 13.0.

```
result asFloat
```

⟹*13.0*

Om vi vill kan vi använda samma kod som tidigare men utföra beräkningarna med flyttal direkt genom att helt enkelt binda h till ett flyttal.

```
dx
    value: [:x | x ** 3  + x + 5]
    value: 2
    value: 1.0e-6
```

⟹*12.3978*

Svaret blir i detta fall 12.3978 och inte 13, efteresom flyttal är definierade med ändlig precision. Men som vi tidigare såg så förlorar vi ingen precision om vi använder rationella tal i beräkningarna till priset av att beräkningarna tar längre tid. Ytterligare ett annat alternativ för att behålla precision är att använda ett tal av typen FixedPoint.

Exempel: Andraderivata

Vi kan enkelt utnyttja det tidigare deriveringsblocket (dx) och konstruera ett block som med en rekursiv teknik beräknar andraderivatan av en funktion given som ett block.

```
d2x := [:f :x :h |
    dx
        value: [:xValue | dx value: f value: xValue value: h]
        value: x
        value: h].
```

och så prövar vi

```
d2 := d2x
    value: [:x | x * 3 + x + 5]
    value: 2
    value: 1 / 1000000.
d2 asFloat
```

⟹*12.0*

Självreferens

Ett block kan också fås att referera till sig själv. Därigenom kan vi skriva rekursiva block.

Exempel: Rekursivt block

Fakultetsfunktionen kan definieras på följande sätt:

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & \text{annars} \end{cases}$$

där n heltal.

Vilket vi enkelt kan åstadkomma med hjälp av ett rekursivt block.

```
fact := [:n| n = 1
            ifTrue:[1]
            ifFalse:[n * (fact value: n -1)]].
```

Vi prövar fact

```
fact value: 6
```

⟹*720*

Hur många argument vill blocket ha?

Vi har hittills tittat på hur man kan utföra block med olika många argument med meddelandet value, value:, etc samt den mer generella valueWithArguments:.

Ibland behöver man dock ta reda på antalet argument som ett block förväntar sig vilket görs med meddelandet numArgs med.

```
[] numArgs
```

⟹*0*

```
[:x0 :y0 :x1 :y1 | x0@y0 corner: x1@y1] numArgs
```

⟹*4*

Block som argument till metoder

Då block är första ordningens objekt och argumenten till metoder kan vara godtyckliga objekt så är det inte heller konstigt att använda block som argument till metoder som vi definierar själva.

För att illustrera det hela konstruerar vi en enkel (lite urartad) klass.

```
Object subclass: #UppEllerNer
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Block-som-argument'
```

Vi definierar en klassmetod som förväntar sig två block med en formell parameter var. Det första blocket (incBlock) ska vara ett block som förväntas ändra värdet av det tredje argumentet (aNumber) ända tills dess att det andra blocket (stopBlock) signalerar att det hela är klart genom en kontroll av aktuellt värde. Slutligen returneras slutvärdet av den förändrade varibeln.

```
incBlock: incBlock stopConditionBlock: stopBlock startValue: aNumber
    | result |
    "Vi kan inte förändra en formell parameter därför använder vi en temporär
    variabel (result)"
    result := aNumber.

    "Vi itererar ända tills stopBlock med result som argument returnerar false"
    [stopBlock value: result]
        "Inne i slingan använder vi incBlock med result som argument"
        whileFalse: [result := incBlock value: result].
    ^result
```

Nu prövar vi det hela, den här gången tilldelar vi som synes argumentblocken till temporära variabler:

```
| incBlock stopBlock |
incBlock := [:aValue | aValue + 5].
stopBlock := [:aValue | aValue \\ 7 = 0].
UppEllerNer
    incBlock: incBlock
    stopConditionBlock: stopBlock
    startValue: 17
```

⇒*42*

Nu byter vi ut incBlock mot ett block som istället ”räknar neråt” och ger detta block direkt som argument till metoden.

```
UppEllerNer
    incBlock: [:aValue | aValue - 5]
    stopConditionBlock: stopBlock
    startValue: 17
```

⇒*7*

Varken argumentet till incBlock: eller till startValue: behöver vara ett tal. Detta illustreras vi med följande anrop:

```
UppEllerNer
    incBlock: [:str | str, str]
    stopConditionBlock: [:x | x size > 25]
    startValue: 'hej'
```

⟹*'hejhejhejhejhejhejhejhejhejhejhejhejhej'*

Att koden kan användas på detta dynamiska sätt beror på Smalltalks otypade natur. Vilket har både sina för- och nackdelar.

Ett blocks omgivning

Ett block har en lexikal (statisk) bindning till sin omgivning. Beroende av hur kodkroppen eller vilka typer av objekt som används i ett blocks omgivning kan ett block delas in i tre olika omgivningstyper.

1 rent
   utan referenser till yttre omgivning och utan kopierad variabellista

2 kopierande
   inga direkta referenser till yttre omgivning. Håller en lista med objekt som används från yttre omgivning. Objekt från yttre omgivning som ej är muterbara kopieras till listan. Om något objekt i blocket fordrar en omgivningspekare kommer även de andra använda den istället för listan

3 fullständig
   håller referens till omslutande omgivning. Objektlistan kommer alltid vara tom (nil) då yttre objekt kan refereras via omgivningspekaren.

Så i följande exempel kopieras z's värde in i blocket:

```
z := 1.
block := [:x :y | x + y + z].
block value: 3 value: 5
```

Men ser koden ut så här istället:

```
z := 1.
block := [:x :y | x + y + z].
z := 2.
block value: 3 value: 5 8
```

eller så här

```
z := 0.
block := [z := z + 1].
block value
```


så konstrueras en omgivningspekare från blocket. Normalt behöver man som programmerare inte bekymra sig om vilken typ som verkligen används.

Felhantering och säker kod

I vissa situationer önskas en hög grad av säkerhet eller garantier för att viss kod alltid utförs även om tidigare kod resulterar i ett felavbrott. Det röra sig om så triviala ting som att återställa cursorns utseende eller mer kritiska som att stänga en fil eller port (eng socket) även om något går snett på vägen.

För sådana ändamål finns två olika metoder definierade, nämligen:

1 kodBlock valueOnUnwindDo: felhanteringsBlock
utför mottagande kodblock men om något går fel så ska felhanterings-Block utföras

2 kodBlock valueNowOrOnUnwindDo: avslutningsBlock
utför mottagande kodblock och utför alltid avslutningsBlock antingen när kodBlock är klar eller om något går fel i blocket innan dess

Ett typexempel på där valueNowOrOnUnwindDo: lämpar sig är följande:

```
öppna infil.
öppna utfil.
[läs från infil.
manipulera innehållet .
skriv resultatet på utfilen] valueNowOrOnUnwindDo: [stäng filerna]
```

Ett situation där en kombination av båda kan vara lämplig är då vi kommunicerar via en port. Här ska vi först konstruera och öppna porten. Om detta första steg spårar ur, men bara då, vill vi avbryta. Annars börjar vi att kommunicera och slutligen om vi antingen är klara eller något går snett på vägen så stänger vi porten.

```
[skapa och öppna port] valueOnUnwindDo: [stäng port och avbryt].
[kommunicera via porten] valueNowOrOnUnwindDo: [stäng porten]
```

För en utförligare diskussion om felhantering se kapitel 12.

Konstruera block från sträng

Det går att konstruera ett block från en sträng. Detta kan tex användas om vi vill konstruera ett program där användaren i dialog ger en funktion. Internt kan det vara bekvämt och effektivt att hantera den som ett

block. Följande exempel utan vidare kommentarer får illustrera det hela.

```
| b |
b := BlockClosure readFromString: '[:x | x squared]'.
b value: 10
```

⟹*100*

```
| user func |
user := 'x sin ** 2'.
func := BlockClosure readFromString: '[:x | ', user, ']'.
func value: 0.5
```

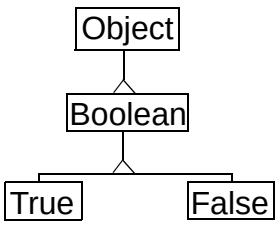
⟹*0.229849*

Block och lättviktsprocesser

Block spelar också en central roll för lättviktsprocesser. Vi beskriver det närmare i kapitel 13.

6.2 Sant eller falskt

Klasserna True och False är de klasser som beskriver metoder för hantering av logiska uttryck och vissa styrstrukturer som villkorsatser.



Gränssnittet, dvs de meddelanden som de ska förstå, är lika. Därför är båda subklasser till den abstrakta klassen Boolean.

Kort beskrivning av klasserna

Då dessa klasser har dragit nytta av Smalltalks totalt objektorienterade struktur, där i princip allt är objekt, blir deras beskrivning och metoder väldigt enkla och rakt fram.

Boolean

En abstrakt klass som definerar klasserna True och False's externa gränssnitt. Här definieras också de metoder som kan beskrivas oberoende av de konkreta subklasserna.

True

Konkret subklass till Boolean. Ska ej instansieras av användare utan dess enda instans är pseudovariabeln true. Här beskrivs också metoderna för de meddelanden som denna instans förstår.

False

False är den andra konkreta subklassen till Boolean. Ska inte heller instansieras av användare utan dess enda instans är pseudovariabeln false. Precis som True definierar vad som ska hända då ett meddelande som skickas till ett uttryck som resulterar i true beskrivs här vad som ska hända om uttrycket istället blir falskt, dvs en instans av false.

Villkorssatser

För att ytterligare illustrera hur enkelt de olika styrstrukturerna kan konstrueras så beskriver vi några av de centrala metoderna i deras klasser. De villkorsatser som finns är ifTrue:, ifFalse: och de som definieras av de kombinerade meddelandena ifTrue:ifFalse: samt ifFalse:ifTrue:. Nu ska vi titta på hur True och False implementerar dem. Gemensamt för dem alla är att de förväntar sig block utan parametrar som argument.

- booleskUttryck ifTrue: omSantBlock

  Det mottagande objektet är true om ”man hamnar i” metoden ifTrue: i klassen True och det är false om vi har hamnat i klassen False. Detta gör att metoderna väldigt enkla att konstruera.

  I True vet vi ju som sagt att villkoret är uppfyllt och att argumentblocket alltid ska utföras.

  ```
  ifTrue: alternativeBlock
      ^alternativeBlock value
  ```

  I False vet vi av samma anledning att blocket aldrig ska utföras och returnerar helt enkelt nil.

  ```
  ifTrue: alternativeBlock
      ^nil
  ```

- booleskUttryck ifFalse: omFalsktBlock

  Metoderna blir helt analoga med ifTrue: fast med ombytta roller mellan True och False.

- booleskUttryck ifTrue: omsantBlock ifFalse: omFalsktBlock

  I True utförs det första blocket och av uppenbara skäl det andra blocket i False.

  True>>ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
  ^trueAlternativeBlock value

  False>>ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
  ^falseAlternativeBlock value

- booleskUttryck ifFalse: omFalsktBlock ifTrue: omsantBlock

  Är analog med ifTrue:ifFalse:.

Logiska operationer

För att utföra logiska operationer finns *och*, *eller* samt *icke* definierade. Det finns två olika typer av *och* och *eller*. En lat som bara beräknar argumentet om det behövs och en omedelbar som beräknar argumentet innan anropet.

operation	**omedelbar**	**lat**
och	&	and:
eller	|	or:
icke	not	
ekvivalens	eqv:	
exklusiv eller	xor:	
om		ifTrue:
om inte		ifFalse:

Figur 6.2 Logiska operationer

- not

  Finns som metod i både True och False.

```
True
  not
    ^false
False
  not
    ^true
```

- & och and:

  Om vi tittar på implementationen av logisk *och* för de båda fallen så ser vi skillnaden mellan dem. Vi ser att & förväntar sig ett ”färdigevaluerat” uttryck medan and: förväntar sig ett block som endast utförs om mottagande objekt är true.

  I True ser metoderna ut så här:

  ```
  & aBooleanValue
      ^aBooleanValue

  and: alternativeBlock
      ^alternativeBlock value
  ```

  I klassen False, där det totala villkoret alltid är falskt, så ser metoderna ut så här:

  ```
  & aBooleanValue
      ^false

  and: alternativeBlock
      ^false
  ```

Övriga metoder för logiska operationer är i princip likadana. Se vidare i Smalltalksystemet.

Utvidgningar av de logiska klasserna

För att illustrera tankegången, fördjupa förståelsen samt konkretisera det hela så ska vi beskriva hur vi kan konstruera egna metoder i de lokiska klasserna.

Låt oss anta att någon önskar villkorsatser med mer ”familjära namn”, för den som har erfarenhet av ett språk ur Algoltraditionen än dem systemet erbjuder.

Först vill vi skriva en metod som heter then:else: som ska vara helt ekvivalent med ifTrue:ifFalse: fast med mer Algolliknande namn på nyckelorden. Följande exempel illustrerar dess användning:

```
x <= y then: ['x mindre än eller lika med y'] else: ['x större än y']
```

Vi önskar också en metod där vi direkt kan ge ett nytt villkor som testas om inte det första (dvs mottagaren) är sant och om detta andra villkor

är uppfyllt ska ett tredje block utföras annars utför vi det fjärde och sista blocket. Typiskt ser dess användning ut som följer:

```
x < y
    then: ['x mindre än y']
    elseIf: [x = y]
    then: ['x och y lika']
    else: ['x större än y']
```

Vi illustrerar två olika möjliga sätt att implementera dessa metoder. Det är en smaksak vilken som väljs. Vi vill påpeka att ändringar av systemklasserna bör man endast göra om man kan motivera dem väl!

Variant: 1, ändringar i True och False

Vi börjar med det enklare fallet och skriver metoden then:else:.

I True vet vi att villkoret var sant och att (precis som vid ifTrue:) det första blocket ska utföras, och metoden kan skrivas på följande sätt:

```
then: trueBlock else: elseBlock
    ^trueBlock value
```

I False ska av motsvarande skäl det andra blocket utföras, dvs:

```
then: trueBlock else: elseBlock
    ^elseBlock value
```

I True blir den andra metoden lika enkel som den föregående (då vi ju vet att mottagaren är sann).

```
then: trueBlock elseIf: otherCondition then: elseIfBlock else: elseBlock
    ^trueBlock value
```

I False blir det lite besvärligare då vi först måste undersöka om det andra villkoret är uppfyllt innan vi väljer vilket block som ska utföras.

```
then: trueBlock elseIf: otherCondition then: elseIfBlock else: elseBlock
    ^otherCondition value then: elseIfBlock else: elseBlock
```

Variant: 2, endast ändringar i Boolean

Om vi vill kan vi istället skriva båda metoderna i klassen Boolean. Här förlitar vi oss på att ifTrue:ifFalse: redan finns.

```
then: trueBlock else: elseBlock
    ^self ifTrue: trueBlock ifFalse: elseBlock

then: trueBlock elseIf: otherCondition then: elseIfBlock else: elseBlock
    ^self
        then: trueBlock
        else: [otherCondition value
                    then: elseIfBlock
                    else: elseBlock]
```

6.3 Odefinierade objekt

Då en variabel skapas så har vi sett att den initialt är nil. Men vad är då nil? Ett objekt! Variabeln nil är ett objekt ur klassen UndefinedObject. Klassen beskriver som vanligt beteendet för sina instanser, i det här fallet nil. UndefinedObject är direkt subklass till Object, därmed förstår nil bla alla meddelanden som är definierade i Object.

Variabeln nil

UndefinedObject instansieras aldrig utan nil är en i systemet given instans. Denna ”variabel” representerar som i många andra språk ett icke-värde, den tomma pekaren eller helt enkelt avsaknaden av något värde. Det ska bara finnas en instans nil. Försöker man ändå skapa ytterligare en instans av UndefinedObject så resulterar detta i ett felavbrott.

Eftersom det bara finns ett objekt ur UndefinedObject, så kommer alla variabler som pekar på nil att peka på samma objekt. Två nyskapade variabler är alltid lika och identiska, dvs

```
| a b |
a == b
```

⇒ *true*

Metoderna isNil och notNil

Om vi exempelvis tittar på isNil och notNil, så framträder återigen Smalltalks totalt objektorienterade struktur fram. Dessa meddelanden har metoder i Object som ser ut som:

```
isNil
    ^false
notNil
    ^true
```

I UndefinedObject ser de däremot ut som:

```
isNil
    ^true
notNil
    ^false
```

Varför en egen klass?

Den främsta orsaken till att en särskild klass för nil har konstruerats är att på den på ett tydligt sätt ska ta hand om fel. Det typiska felet i Smalltalk är att ett meddelande skickas till ett objekt som inte förstår det. Ibland beror deta på att en variabel inte är initierad på ett riktigt sätt och den har då ofta värdet nil.

Det hade varit möjligt att definiera nil som instans av tex Object men då hade felmeddelanden förorsakade av att mottagaren är nil blivit mindre tydliga. Vi hade då fått ett felmeddelande i stil med

Object förstår ej meddelandet meddelandenamn

nu får vi

UndefinedObject förstår ej meddelandet meddelandenamn

vilket är tydligare eftersom det framgår att mottagaren är odefinierad.

6.4 Metaklasser

I Smalltalk gäller följande:

- alla komponenter är objekt
- alla objekt är instanser av klasser

Detta leder till att alla klasser är objekt. Vilket i sin tur leder till att alla klasser är instanser av andra klasser.

En klass vars instanser är klasser kallas för en *metaklass*.

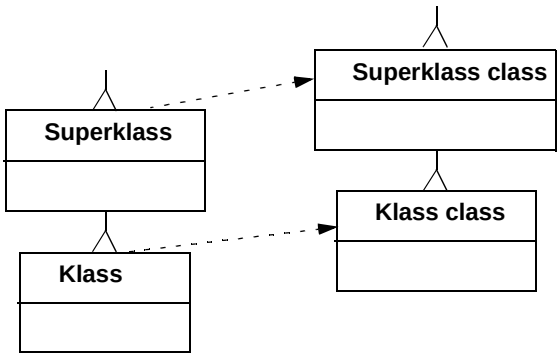
Kort historik

I tidiga versioner av Smalltalk så fanns endast en metaklass, som hade namnet Class. Ett problem med denna konstruktion var att alla klasser delade en given mängd konstruktörer, som var definierade på ett och samma ställe. Detta medförde att inte några speciella, klassberoende, initieringar kunde göras vid instansieringen av dem.

Genom att istället göra varje klass till instans av sin egen metaklass försvann detta problem.

Instanser av metaklasser

En metaklass har inget eget namn utan refereras genom att skicka meddelandet class till klassen. En metaklass är en klass så den har en superklass. Om den vanliga klassen har superklass så är den instans av den tidigare klassens metaklass superklass, vilket följande figur illustrerar:



Figur 6.3 Klass och metaklass

Exempel: Metaklass

Anta att vi har följande instans av Point: p := 10@20.

a) Hur kan vi från p ta reda på dess klass

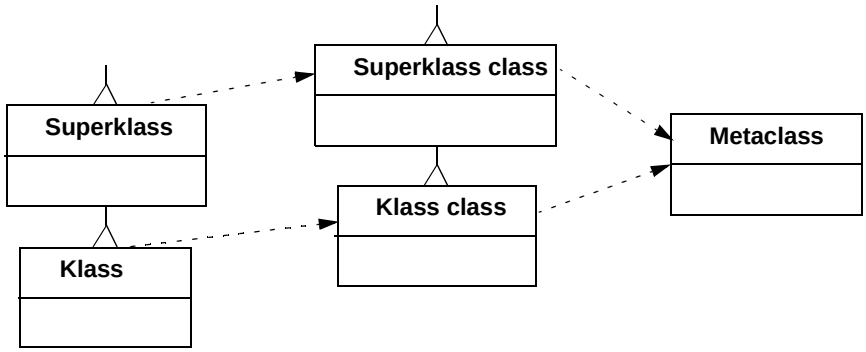
b) Hur får vi reda på metaklassen

Lösning: Använd meddelandet class i båda fallen

a) klassen := p class

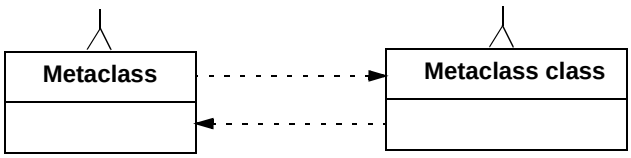
b) metaklassen := klassen class

Alla metaklasser är instanser av klassen Metaclass, vilket illustreras i figur 6.4.



Figur 6.4 Metaklasser är instanser av Metaclass

Då frågar man sig kanske vad klassen Metaclass är för något. Svaret är att den (självklart) är en vanlig klass, som i sin tur är instans av klassen Metaclass class, dvs av en metaklass. Regeln om att alla metaklasser är instanser av Metaclass gör sedan att Metaclass class är instans av Metaclass!



Figur 6.5 Metaclass är instans av Metaclass class och vice versa

Metaklassernas hierarki

Nu återstår bara att reda ut vad som är superklasser till de olika metaklasserna. Regeln är ju den att Object är superklass till alla klasser. Hur inbegrips då metaklasserna i denna regel?

I diskussionen om metaklasser i avsnitt 4.2 så sa vi att Behavior beskriver en klass relation i klasshierarkin, dess metoder samt dem grundläggande konstruktörerna new och new:. Klassen ClassDescription ansvarar för metoder för att hantera klasskommentarer, klassens kategori och instansvariabler. Klassen Class adderar mer heltäckande programmeringsverktyg till Behavior. Speciellt finns metoderna för att skapa nya subklasser och hanteringen av klass- och poolvariabler här. Metaclass tillhandahåller metoder för initiering av klassvariabler samt av nya instanser av de vanliga klasserna (som är instanser av Metaclass).

Klasshierarkin för Behavior med subklasser, med instansvariabler inom parentes, ser ut som följer:

```
Object ()
    Behavior ('superclass' 'methodDict' 'format' 'subclasses')
        ClassDescription ('instanceVariables' 'organization')
            Class ('name' 'classPool' 'sharedPools')
                ... alla metaklasserna ...
            Metaclass ('thisClass')
```

Som ni ser har vi här fått in Metaclass på sin plats i arvsträdet, som subklass till ClassDescription. Vidare ser vi att den rätta platsen att placera in metaklasserna är som subklasser till klassen Class. Dvs här sker en ”övergång” från metaklasser till ”vanliga” klasser som i sin tur medför att Object är slutgiltig superklass till dessa också.

Nu ska vi rita en graf över hur klasser och metaklasser relateras (figur 6.7 nedan). I grafen har vi, förutom de mest centrala klasserna i detta sammanhang, också lagt in Point och dess relationer.

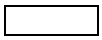
För att förtydliga vad som är vad i figuren har vi valt att använda en grafisk notation som beskrivs i figur 6.6.

Vi har valt notationen för att göra det enklare att läsa figuren samt förtydliga skillnader (som egentligen inte är några skillnader).

Kategorin Object Kernel

Vi avslutar detta avsnitt om metaklasser med att diskutera lite mer konkret vad deras existens leder till samt hur klasserna som hanterar dem ser ut.

Klassen Behavior beskriver vad som är nödvändigt för objekt som har instanser. Detta inkluderar hantering av klassvariabler, metodkataloger, skapande av instanser samt hantering av initiering av en viss klass. I klassen beskrivs också grundläggande gränssnitt mot kompilatorn.

Typ av objekt	Grafisk presentation
Vanlig klass	
Metaklass	
Arv mellan vanliga klasser	
Arv mellan metaklasser	
Arv från vanlig klass till metaklass	
Instans av vanlig klass	
Instans av metaklass	

Figur 6.6 Grafisk notation i figur 6.7

Vi kan sammanfatta Behavior's viktigaste ansvarsområde och beteende i följande punkter med instansmetoder:

1 skapa instans
Metoderna new, new:, basicNew, basicNew: är implementerade (i sina grundutföranden) här.

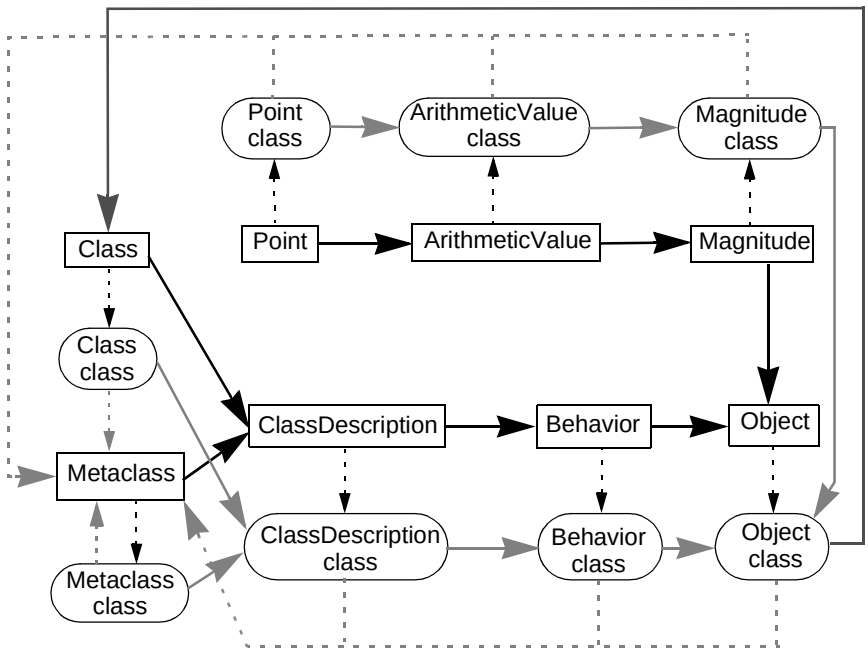
2 skapa metoder
Metoder för att lägga till, ta bort och gränssnitt mot kompilatorn för att kompilera kod finns här.

3 klasshierarkier
Metoder för att hantera en klass pekare till superklass och subklasser finns här. Det finns också metoder för att ta bort subklasser.

4 hantera kod
Möjligheter att hitta metodnamn som förstås av klassen och var i hierarkin de är implementerade, källkod samt metoddefinitioner hanteras av metoder i Behavior.

5 instanser
Klassen ger oss också möjligheter att hitta och skicka meddelanden till alla instanser, superklasser eller subklasser av en klass.

Klassen ClassDescription lägger till en del faciliteter till den basala beskrivningen och hanteringen som görs av Behavior. Bland annat han-



Figur 6.7 Klassernas och metaklassernas relationer

teras namngivna instansvariabler, klasskommentarer och en del smidigare sätt att kompilera kod eller kopiera metoder från andra klasser här.

Klassen Class ger ytterligare stöd för kodutveckling. Bla definieras vilken kompilator som ska användas för den aktuella klassen, metoderna för att skapa subklasser (som tex används i browsern) samt flera metoder för att hantera klasser.

Klassen Metaclass tillhandahåller främst metoder för att initiera klasser.

6.5 Meddelanden, metoder, exekvering och omgivning som objekt

Vi ska nu titta lite närmare på hur de mest grundläggande delarna i ett objekts hantering och meddelandesändning ser ut. Vi kommer beskriva det hela ur Smalltalksomgivningens synvinkel och alltså inte beskriva detaljer i den virtuella maskinen.

Metoduppslagning

Man kan betrakta en metod på ett antal olika sätt, tex som källkod eller som kompilerad kod. Gemensamt är att varje klass hanterar en katalog med metodnamnen som nycklar och dess kod som värden.

Metodnamn

Vi kan få reda på namnet för alla metoder som är definierade i en viss klass genom att skicka meddelandet selectors till klassen.

Om vi på en gång vill får reda på alla metodnamn som antingen är definierade i den mottagande klassen eller någon av dess superklasser så kan vi använda allSelectors.

För att ta reda på var någonstans i klassträdet närmaste metod med givet namn finns använder vi findSelector:.

Förstår objekt ur klassen ett visst meddelande?

Vi har redan tidigare sett prov på användningen av respondsTo: som ger oss svar på frågan om mottagaren förstår ett meddelande. Vad som sker vid denna fråga är att klassen konsulteras och undersöker om ett meddelande med detta namn finns i dess metodkatalog. Om så är fallet så returneras true. Om inte aktuellt metodnamn hittas forsätter sökningen i superklassen ända tills någon klass metodkatalog inkluderar namnet eller mottagande klass superklass är nil, då returneras false.

Metoddefinition

Hantering av en klass metoder och kod sker också i klasserna Class med superklasser. Det vanligaste är att man vill komma åt en metods källkod (fast detta görs ju oftast från browsern). Ibland kan man också vilja hantera eller undersöka en metods kompilerade kod.

- Källkod

  Källkoden för en viss metod får vi genom att skicka sourceCodeAt:, med metodnamnet som argument, till den klass där metoden finns.

- Kompilerad kod

  Den kompilerade koden för en viss metod får vi genom att skicka compiledMethodAt:, med metodnamnet som argument, till den klass där metoden finns.

Exekvering av metoder beskrivna som objekt

Det är inte så ofta som man behöver behandla meddelanden eller metoder som objekt. En vanlig situationen då detta ändå inträffar är vid konstruktion av pop-up-menyer och liknande. Det är också bra att känna till klasserna Message med subklasser, bla då meddelanden konverteras till instanser av dessa vid vissa felavbrott. Om man vill hantera en metods relationer och omgivning behöver man också känna till dem.

Metodnamn given i variabel

Ibland kan det hända att man inte direkt då koden skrivs vet vilken metod som ska användas. I sådana situationer kan man istället lagra ett metodnamn i en variabel. För att skicka ett meddelande som antingen direkt är given på symbolisk form eller via en symbol i en variabel används perform:.

```
12.56 perform: #fractionPart
```
⟹*0.56*

Om meddelandet också kräver ett argument så används istället perform:with:.

Exempel: Perform med argument

```
| operation receiver argument |
operation := #max:.
receiver := 10.
argument := 4 factorial.
receiver perform: operation with: argument.
```
⟹*24*

I figur 6.8 är de olika perform-meddelandena beskrivna:

Meddelande	Kommentar
perfom:	Unär metod.
perform:with::	Ett argument.
perform:with:with:	Två argument.
perform:with:with:with:	Tre argument.
perform:withArguments:	Noll eller flera argument. Argumentet till withArguments: ska vara en Array med rätt antal parametrar.

Figur 6.8 Perform-meddelanden

Exempel: Perform med argumentvektor

```
| op receiver |
op := #findString:startingAt:ignoreCase:useWildcards:.
receiver := 'Det går enkelt att söka i strängar om man känner till dom rätta
metoderna!'.
receiver
    perform: op
    withArguments: (Array with: 'r###a'
                               with: receiver size // 2
                               with: false
                               with: true)
```

⟹*(59 to: 63)*

Hur många argument tar ett meddelande emot?

Om man vill kan man ta reda på hur många argument ett nyckelord givet av en symbol har:

```
#between:and: numArgs
```

⟹*2*

Det går också att få nyckelorden ”uppstyckade” och levererade i en vektor:

```
#between:and: keywords
```

⟹*#('between:' 'and:')*

Speciella meddelandeobjekt

Klassen Message håller i ett meddelande i form av dess metodnamn och argument.


```
| m |
m := Message selector: #between:and: arguments: #(5 10).
3 perform: m selector withArguments: m arguments.
```

⇒*false*

Om man på ett smidigare sätt vill utföra koden och ange en mottagare till meddelandet kan Message's subklass MessageSend användas.

```
| m |
m := MessageSend selector: #between:and: arguments: #(5 10).
m receiver: 3.
m value.
```

⇒*false*

Nu kan vi enkelt återanvända meddelandet och argumenten trots att vi byter mottagare.

```
m receiver: 7.
m value
```

⇒*true*

Kompilerade metoder

Det går som vi såg att få tag ett objekt som hanterar en metods kompilerade kod. Vi får tag i en metods kompilerade kodobjekt genom att använda compiledMethod: med aktuellt metodnamn som argument. En sådan unär metod kan exekveras direkt genom att performMethod:, med det kompilerade kodobjektet som argument, skickas till ett mottagande objekt. Om metoden däremot kräver argument så används istället performMethod:with:, performMethod:with:with: osv (se Object).

Primitiva metoder

Ibland träffar man på kod i stil med följande i en metod:

```
<primitive: 569>
```

Denna kod står alltid direkt efter metodhuvudet och kan efterföljas av vanliga Smalltalksatser. Detta är en uppmaning till den virtuella maskinen att direkt utföra ett meddelande. Om det misslyckas, och bara då, utförs den efterföljande Smalltalkkoden. Detta är ett sätt att göra koden maximalt effektiv, samtidigt som den är flyttbar mellan datorer.

Felmeddelanden som metoder och objekt

Om ett objekt inte förstår ett meddelande så skickas meddelandet doesNotUnderstand: till det. Argumentet är en instans av Message som innehåller namnet på den metod samt argument till den metod som spårade ur.

Metoden doesNotUnderstand:

I klassen Object är metoden definierad på följande sätt:

```
doesNotUnderstand: aMessage
    | selectorString |
    selectorString := Object errorSignal
        handle: [:ex | ex returnWith: '** unprintable selector **']
        do: [aMessage selector printString].
    Object messageNotUnderstoodSignal
        raiseRequestWith: aMessage
        errorString: 'Message not understood: ' , selectorString.
    ^self perform: aMessage selector withArguments: aMessage arguments
```

Vid anrop av metoden visar sig ett felfönster. Från detta fönster kan man avbryta, avlusa eller direkt försöka att forsätta exekveringen. Det senare kan tex ske efter att metoden som fallerade definierats på konventionellt sätt i en browser.

Exempel med "mjukis"

För att illustrera hur en klass kan definiera sitt eget beteende vid fel konstruerar vi en liten klass som istället för att ge ett felavbrott, om en viss metod inte förstås, skriver ut lite information om det hela i utskriftsfönstret.

```
Object subclass: #Mjukis
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Felhantering'
```

Vi skriver om metoden doesNotUnderstand: på följande sätt:

```
doesNotUnderstand: aMessage
    Transcript show: 'Jag förstår inte (', aMessage printString, ')'; cr
```

Nu kan vi prova med ett meddelande som inte instanserna ska förstå.

```
mjukis := Mjukis new.
mjukis radius: 10 angle: 25
```

⟹*I utskriftsfönstret skrivs:*
Jag förstår inte (a Message with selector: #radius:angle: and arguments: #(10 25))

Nu ska vi ändra koden så att felavbrottet blir det som är definierat i Object om ett visst felaktigt meddelande skickas till en instans fler än tre gånger.

Vi definierar en instansvariabel tracer och ser på vanligt sett till att den blir initierad genom att definiera klassmetoden:

```
new
    ^super new initialize
```

och instansmetoden:

```
initialize
    tracer := Bag new
```

En Bag är en behållare med en räknare för hur många gånger var och en av de objekt som stoppas i den förekommer.

```
doesNotUnderstand: aMessage
    (tracer occurrencesOf: aMessage selector) >= 3
        ifTrue: [^super doesNotUnderstand: aMessage].
    tracer add: aMessage selector.
    Transcript show: 'Jag förstår inte (' , aMessage printString , ')'; cr
```

Så att tex följande händer:

```
mjukis reciprocal.    Mjukt fel
mjukis + 3.           Mjukt fel
mjukis reciprocal.    Mjukt fel
mjukis + 5.           Mjukt fel
mjukis reciprocal.    Mjukt fel
mjukis reciprocal.    Hårt fel
mjukis + 7.           Mjukt fel
mjukis + 9.           Hårt fel
```

där *Mjukt fel* indikerar att endast felmeddelandet med utmatning i utskriftsfönstret utförs och *Hårt fel* att ett konventionellt felavbrott sker.

6.6 Roten i arvsträdet

Hittills har vi alltid sagt att alla klasser ärver från Object, och så kommer vi nog säga i fortsättningen också. För trots allt så befinner sig *så gott som alla* klasser i en gren av ett träd med Object som rot.

Det är det där lilla *så gott som* som vi ska diskutera nu. Det går nämligen att definiera en klass utan att Object är dess rot!

Hur definierar jag en annan rot än Object?

Om vi istället för att ange en klass som superklass skriver nil så kommer den nya klassen inte få någon superklass.

```
nil subclass: #MinEgenRot
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Ny rot'
```

Alternativt kan vi efter det vi har definierat en klass på vanligt sätt gå in och hantera pekare (tex superclass) som hanteras av metaklassen. Vi fördjupar oss inte i detta.

Vad händer när vi definierar en ny rot?

Först varnar systemet om att en klass med nil som superklass håller på att definieras. Om man väljer att forsätta ändå så konstrueras klassen samt de två metoderna class och doesNotUnderstand:. Den första av dessa metoder blir helt enkelt ett primitivanrop. Den andra är intressantare.

```
doesNotUnderstand: aMessage
    (Object canUnderstand: aMessage selector)
        ifTrue:[self class copy: aMessage selector from: Object.
                ^self perform: aMessage selector
                    withArguments: aMessage arguments].

    ^Object messageNotUnderstoodSignal
        raiseRequestWith: aMessage
        errorString: 'Message not understood: ' , aMessage selector
```

Metoden finns inte i MinEgenRot men om den finns i Object så kopieras den och sedan utförs meddelandesändningen igen!

Metoden class måste finnas så den får vi inte ta bort. Däremot kan vi ändra beteendet för doesNotUnderstand:, så att vi *verkligen* får nil som superklass. Något vi däremot inte kan ändra, hur vi än bär oss åt, är att meddelandet == kommer förstås av instanserna av klassen. Referenslikhet går inte att definiera om.

Vilka problem finns det med en egen rotklass?

Ett problem är att vi *inte får* ett objekt utan superklass på riktigt. Detta då metoder som inte finns i det mottagande objektets klass men i Object kopieras. En strategi för att lösa detta är att modifiera metoden och ta bort den del som kopierar och utför meddelandet igen (dvs de första fyra raderna av kodkroppen). En bättre strategi är troligen att kopiera meto-

den perform:withArguments: och doesNotUnderstand: från Object. Där den första behövs då den anropas av den andra.

Ett annat problem är att om vi har sparat klassen på sekundärminne så kommer systemet varna och stoppa upp om vi läser in denna kod med **file in**. Fast vi kan fortsätta!

Varför skulle man vilja ha en annan rot än Object?

Det kan ju hända att man vill konstruera ett objekt som inte förstår alla meddelanden som finns i Object eller konstruera en egen rotklass där man kan härja fritt. Men den vanligaste situationen är då man programmerar med ersättare eller inkapslingsobjekt, speciellt vid transperent kommunikation med filer, databaser eller i distribuerade system. Vi återkommer till detta i avsnittet ”Programmering med ersättare" nedan.

6.7 Syntaxanalysator, Scanner och Kompilator

När man skriver in Smalltalkkod i en browser och man väljer **accept** så kommer den att kompileras av en kompilator som själv är skriven i Smalltalk. Det betyder att det är fullt möjligt att förändra Smalltalks syntax genom att förändra syntaxanalysatorn eller kompilatorn. Detta understödjs faktiskt av systemet på så sätt att varje metaklass avgör vilken kompilator som ska användas för just den här klassen. Därigenom är det möjligt att för vissa klasser utöka eller förändra syntaxen om det behövs. Ett programpaket för att anropa egendefinierade primitiver, från ParcPlace-systems bygger just på det. Där är syntaxanalysatorn förändrad så att det är möjligt att vid anrop av primitiv ange primitivnamn och parametrar direkt enligt ANSI-C standard i en vanlig Smalltalk-browser.

6.8 Klassinstanser och objektidentitet

Alla objekt har en objektidentitet som ges automatiskt till dem vid konstruktion. Denna identitet hanteras av den virtuella maskinen och är ej påverkbar från Smalltalkkoden. Däremot kan man få objekt att peka ut gemensamma objekt genom en vanlig tilldelning.

Ibland kan det också vara intressant att utan att direkt förfoga över en pekare till en viss klass instanser ändå kunna få tag i dem. I andra situ-

ationer kan det hända att man behöver byta ett objekts klass eller helt enkelt ersätta det med ett annat objekt ur samma eller annan klass.

Vandra i klasshierarkin och bland dess instanser

Vi ger en liten provkarta på metoder som hanterar en klass hierarki och dess instanser. För att hitta fler metoder och mer i detalj utforska det hela hänvisar vi till klassen Behavior med subklasser.

Vandra i klasshierarkin

I tabell 6.9 nedan beskriver vi några av de viktigaste metoderna för att hantera en klass hierarki.

Metod	**beskrivning**
superclass	mottagarens superklass
subclasses	en lista av alla mottagarens direkta subklasser
allSubclasses	alla subklasser ända ner till löven, med bredden först.
allSuperclasses	alla superklasser upp till roten
withAllSuperclasses	alla superklasser och mottagaren
withAllSubclasses	allSubclasses och mottagare
commonSuperclass:	vilken är mottagarens och argumentets närmaste gemensamma superklass?
whichClassIncludesSelector:	vilken är den närmaste superklassen som implementerar en metod med namn givet i argumentet?

Figur 6.9 Hitta klasser i hierarkin

Vandra bland instanserna

I tabell 6.10 nedan beskriver vi några av de viktigaste metoderna för att hantera en klass instanser, speciellt om vi inte har några pekare till dem.

Metod	beskrivning
allInstances	alla instanser av klassen
instanceCount	hur många instanser har klassen
instVarNames	vilka instansvariabler finns
classVarNames	vilka är klassvariablerna
someInstance	ge mig en godtycklig instans av klassen
subclassInstVarNames	instansvariabelnamnen i de direkta subklasserna

Figur 6.10 Hitta instanser i hierarkin

Operationer på klasser eller instanser i hierarkin

I tabell 6.11 nedan beskriver vi några metoder för att utföra ett visst block på alla instaser av en viss klass eller dess subklasser..

Metod	beskrivning
allInstancesDo:	utför argumentblocket på alla klassens instanser. Argumentblocket ska ha en formell parameter som representerar aktuell instans.
allSubclassesDo:	utför argumentblocket på alla klassens subklasser. Argumentblocket ska ha en formell parameter som representerar aktuell subklass.
allSubInstancesDo:	utför argumentblocket på alla subklassernas instanser. Argumentblocket ska ha en formell parameter som representerar aktuell instans.

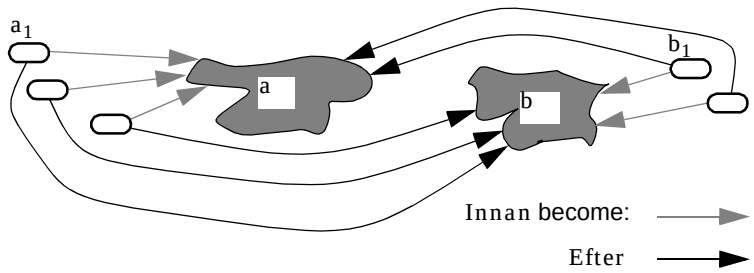
Figur 6.11 Operera på instanser eller klasser i hierarkin

Ändra ett objekts identitet

I vissa situationer kan det vara av intresse att ändra ett objekts identitet. Lämpligast kan detta åstadkommas genom att struktur och design redan från början är förberedd för detta. I vissa situationer kan det vara bättre att byta ett visst objekts identitet mot en annan. I Smalltalk finns, till skillnad från de flesta andra programmeringsspråk, stöd för detta!

Meddelandet become:

Meddelandet become: används om man önskar att det objekt som en viss



Figur 6.12 a_1 become: b_1, i Smalltalk-80

variabel pekar på ska bytas mot ett annat objekt; samtidigt som alla andra pekare till detta objekt också ska fås att peka på detta andra objekt. I Smalltalk-80 medför detta också att alla pekare till det andra objektet ändras till att peka på det första objektet. I tex Smalltalk\V resulterar detta däremot *inte* i att pekarna till det andra objektet sätts att peka till det första.

I Smalltalk-80 kan man *inte* använda become: med block, metodomgivningar, heltal, tecken eller nil, som mottagare eller argument.

```
a1 := 5@7.
a2 := a1.
a3 := a1.                          alla a refererar till en punkt
b1 := 'En sträng'.
b2 := b1.                          alla b refererar till en sträng
a1 become: b1.                     här byter alla a och b typ
b2 x: 10.                          b2 är numera en punkt
'a1: ', a1, ' a2: ', a2, ' a3: ', a3, ' b1: ', b1 printString, ' b2: ', b2 printString
```

⇒ *'a1: En sträng a2: En sträng a3: En sträng b1: 10@7 b2: 10@7'*

Meddelandet changeToClassOfThat:

En alternativt sätt, där inte två instanser byter plats utan där det mottagande objektets klass byter klasstillhörighet till den klass som argumentet tillhör, är att använda metoden changeToClassOfThat:.

Den här metoden är ännu mer restriktiv än become: då mottagare och argument måste ha precis samma antal instansvariabler.

Programmering med ersättare

En allt vanligare teknik i samband med objektorienterad programmering är att dölja ett objekts specifika identitet eller lokalitet med hjälp av ersättningsobjekt (eng. proxy). Speciellt vanligt är detta i distribuerade omgivningar och system.

Gemensamt för de flesta typer av ersättare är att de ska ha ett så minimalt externt gränssnitt som möjligt, dvs vanligen med nil som superklass, och fånga meddelanden och skicka dem vidare till originalobjektet. Ibland kan det också vara bekvämt att låta en ersättare fullständigt mutera med det objekt den representerar, dvs med become:, så att alla eventuella pekare automatiskt sätts om. Då kan ett från början normalt objekt bli ett objekt som hanteras med ersättare.

Muterande ersättare

För att illustrera användning och konstruktion av ersättare ska vi nu konstruera en som tar ett givet objekts plats. Erättaren ska därefter logga meddelandesändning till objektet och i vilken klass i objektets arvsträd som metoden finns. Efter slutfört arbete skickas ett speciellt meddelande till ersättaren och resultatet returneras.

Vi kommer konstruera en ny klass utan superklass samt använda become: för muteringen. Det senare gör att vi inte kan kapsla in alla typer av objekt, se sidan 222.

Klassdefinitionen ser ut som följer:

```
nil subclass: #TraceProxy
    instanceVariableNames: 'object tracedMethodLookups '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Ersättare'
```

Instansvariabeln object kommer att vara det objekt vars meddelandesändning vi undersöker.

Vi skriver precis som för klassen Mjukis om klassmetoden new med anrop av initialize som i sin tur initierar tracedMethodLookups till att vara en OrderedCollection.

```
new
    ^super new initialize

initialize
    tracedMethodLookups := OrderedCollection new
```

Förutom dessa metoder skriver vi också klassmetoden:

```
on: anObject
    ^self new theEncapsulatedObject: anObject
```

Då vi vill kopiera vissa metoder från Object, även vid inläsning till ny image, så skriver vi följande klassinitieringsmetod.

```
initialize
    self == TraceProxy
        ifTrue:
            [superclass := nil.
            #(#class #basicInspect #basicSize #become: #perform:
            #perform:with: #perform:with:with: #perform:with:with:with:
            #perform:withArguments: #performMethod: #performMethod:with:
            #performMethod:with:with: #performMethod:with:with:with:
            #performMethod:arguments:) do:
                        [:selector | self copy: selector from: Object]]
```

Obeservera särskilt hur vi kontrollerar om mottagaren är förväntad klass. Detta gör att eventuella subklasser undviker att göra om denna för dem onödiga initiering. Vidare sätter vi speciellt superklasspekaren till nil. Detta medför att vi kan spara klassen med en superklass definierad och sedan vid inläsningen ta bort den. Detta gör att irriterande felmeddelanden undviks vid **file in**. Observera också hur vi kopierar metoder som vi behöver från Object (dvs med copy:from:).

Nu skriver vi också metoden som sätter upp relationen mellan ersättaren och originalobjektet.

```
theEncapsulatedObject: anObject
    " Sparar en kopia av objektet så att vi kan rikta meddelanden till det"
    object := anObject shallowCopy.
    anObject become: self                          mutera och byt pekare
```

Observera att vi måste spara en kopia av anObject annars skulle även denna referens mutera vid become:, vilket skulle medföra att vi inte skulle ha någon referens till originalobjektet.

Som vanligt hamnar infångning av meddelanden och logik i doesNotUnderstand:.

```
doesNotUnderstand: aMessage
    | messageImplementor |
    "Vilken klass implementerar metoden som ska utföras?"
    messageImplementor := self theEncapsulatedObject class
                              whichClassIncludesSelector: aMessage selector.
    "Vi lägger in informationen som en association, dvs mha nyckel->värde"
    tracedMethodLookups add: messageImplementor printString , '>>' ->
                                                              aMessage.
    ^self theEncapsulatedObject
        perform: aMessage selector
        withArguments: aMessage arguments
```

När vi inte längre vill använda ersättaren utan återgå till det gamla objektet så återställer vi det hela och returnerar resultatet av loggningen.

```
restoreTheEncapsulatedObjectAndReturnTrace
    | theTrace |
    theTrace := tracedMethodLookups.
    self become: self theEncapsulatedObject.
    ^theTrace
```

Vi testar:

```
point := 10 @ 20.
vh := point asValue.
proxy := TraceProxy new.
proxy theEncapsulatedObject: point.
point x: 100.
Transcript print: point restoreTheEncapsulatedObjectAndReturnTrace; cr.
Transcript print: point; cr.
Transcript print: vh value; endEntry
```

Ger följande i utskriftsfönstret

```
⟹OrderedCollection ('Object>>'->a Message with selector: #myDependents and
arguments: #() 'Point>>'->a Message with selector: #x: and arguments: #(100)
'Object>>'->a Message with selector: #myDependents and arguments: #()
'Object>>'->a Message with selector: #primBecome: and arguments: #(100@20))
100@20
100@20
```

Där de flesta metoderna härrör från become:. Vi gör en test till för att titta lite mer på metoduppslagningsdelen:

```
point := 10 @ 20.
proxy := TraceProxy on: point.
point + (10 @ 10).
point between: 2 @ 3 and: 12 @ 15.
Transcript print: point restoreTheEncapsulatedObjectAndReturnTrace; cr.
Transcript endEntry
```

Ger följande, lite friserade, utmatning

```
⟹'Object>>'->a Message with selector: #myDependents and arguments: #()
'Point>>'->a Message with selector: #+ and arguments: #(10@10)
'Magnitude>>'->a Message with selector: #between:and: and arguments: #(2@3
    12@15)
'Object>>'->a Message with selector: #myDependents and arguments: #()
'Object>>'->a Message with selector: #primBecome: and arguments: #(10@20))
```

Icke muterande ersättare

Det tidigare beskrivna sättet, att låta ett objekt ta rollen av ett annat med byte av pekare mellan dem, fungerar endast under vissa speciella omständigheter och är nog inte lika vanlig som då man endast utnyttjar ett objekt utan superklass. I många fall går det inte heller, eller har ingen mening att använda become:, tex om det objekt man vill kapsla in är ett objekt som man inte tillåts att ”byta identitet” med, en fil eller ett objekt som man kommunicerar med via en port. Vi beskriver inte detta närmare då det i viss mån kan ses som ett specialfall av det muterande fallet.

6.9 Exempel

Repeat until

Vissa programmerare saknar en konstruktion motsvarande Pascal's **REPEAT UNTIL**. Detta är dock inte något större problem, då vi åter igen kan utnyttja att allt är objekt och helt enkelt definiera våra egna styrstrukturer! Första tanken är kanske att skriva något i stil med:

```
repeat: ettBlock until: ettAnnatBlock
```

Men det går inte eftersom meddelandet repeat:until: måste skickas till något objekt. Vi flyttar ettBlock till före repeat och får metoden:

```
repeatUntil1: anotherBlock
    self value.
    ^anotherBlock value ifFalse: [self repeatUntil1: anotherBlock]
```

Nu kan vi direkt pröva våra nya styrstruktur.

```
| i |
i := 0.
[i := i + 1. Transcript  print: i; endEntry] repeatUntil1: [i >= 10]
```

⟹*12345678910*

Den rekursiva beskrivningen av repeatUntil1: ovan ger oss ganska dålig prestanda så vi ger två alternativa lösningar som drar nytta av att while-slingor är optimerade.

```
repeatUntil2: anotherBlock
    self value.
    ^anotherBlock whileFalse: self
```

I det här fallet kommer VisualWorks kompilator klaga över att vi inte har använt litterala block, men det går att fortsätta i alla fall om man vill.

Om vi inte vill få klagomål och dessutom garantera att koden optimeras, kan vi istället implementera det hela enligt följande:

```
repeatUntil3: anotherBlock
    self value.
    ^[anotherBlock value] whileFalse: [self value]
```

där vi uttryckligen klargör att de inblandade objekten är block.

Case-sats

Ibland hör man någon beklaga sig över att Smalltalk inte har någon case-sats! För att illustrera hur man enkelt kan konstruera en dito, om man nu absolut vill ha en så ska vi nu kort skissa hur detta kan gå till.

Inom parentes kan vi påpeka att pop-up-menyer är en form av case-sats och Smalltalk hanterar dem ofta med en ”case-liknande” teknik.

```
| case exp otherwise |
case := Dictionary new
    add: 1 -> [Transcript cr; show: 'måndag'];
    add: 2 -> [Transcript cr; show: 'tisdag'];
    add: 3 -> [Transcript cr; show: 'onsdag'];
    add: 4 -> [Transcript cr; show: 'torsdag'];
    add: 5 -> [Transcript cr; show: 'fredag'];
    add: 6 -> [Transcript cr; show: 'lördag'];
    add: 7 -> [Transcript cr; show: 'söndag'];
    add: #b -> [Transcript cr; show: 'b'];
    yourself.
otherwise := [Transcript cr; show: 'felaktigt dagnummer'].

(case at: 1) value.
```

⇒*måndag*

```
(case at: 99 ifAbsent: [otherwise]) value.
```

⇒*felaktigt dagnummer*

I praktiken, om man vill använda en ”case-sats”, bör den sparas i en klass- eller instansvariabel.

Symbolisk aritmetik och derivering

Nu ska vi visa hur ett betraktelsesätt där vi hanterar matematiska funktioner som objekt är fruktbart vid symbolisk aritmetik och derivering.

Vi kommer att bygga upp en enkel hierarki av klasser som representerar konstanter, summor, produkter, bråk och monom, se figur 6.13. Inom denna grupp av objekt ska det sedan vara möjligt att först utföra enkla aritmetiska operationer och senare symbolisk derivering.

Då avsikten främst är att illustrera tekniker, peka på möjligheter utan att kodvolymen ska bli onödigt stor undviker vi att till att börja med att matematiskt förenkla resultaten av operationerna. Att förenkla ett delresultat är heller inget enkelt problem att lösa. Vad som är enklaste form kan också ofta bero på den omgivning funktionen befinner sig i. I avsnittet ”Förkortning av resultat" på sidan 237 pekar vi på några möjligheter att på ett strukturerat sätt göra sådana förenklingar av resultaten.

För att konstruera det hela kommer vi att bland annat använda oss av en gemensam abstrakt klass i vilken stora delar av koden kan implementeras. Vi kommer basera vår lösning på dubbel uppslagning vilket är en anledning till att stora delar av koden kan hamna i den abstrakta

klassen. Detta banar också väg för senare optimeringar och förenklingar av de matematiska uttrycken. I den abstrakta klassen hamnar också metoder som anger skönsvärden för olika tester och jämförelser.

Klasser, definitioner och grunder

Uttryck	**Deriveringsregel**
Konstant	$\frac{dc}{dx} = 0$
Variabel	$\frac{dx}{dx} = 1$
Monom	$\frac{d}{dx}(a \cdot x^n) = n \cdot a \cdot x^{n-1}$
Summa	$\frac{d}{dx}(f + g) = \frac{df}{dx} + \frac{dg}{dx}$
Produkt	$\frac{d}{dx}(f \cdot g) = \frac{df}{dx} \cdot g + f \cdot \frac{dg}{dx}$
Bråk	$\frac{d}{dx}\left(\frac{f}{g}\right) = \frac{\frac{df}{dx} \cdot g - f \cdot \frac{dg}{dx}}{g^2}$

Figur 6.13 Deriveringsregler

För att förtydliga, och minimera risk för krockar med existerande klasser, inleder vi alla klassnamn med prefixet DF.

Class:	DFAbstract
Superclass:	Object

Class:	DFConstant
Superclass:	DFAbstract
Instance variables:	number

Class:	DFMonom
Superclass:	DFAbstract
Instance variables:	factor symbol exponent

Class:	DFSum
Superclass:	DFAbstract
Instance variables:	termOne termTwo

Class:	DFProduct
Superclass:	DFAbstract
Instance variables:	factorOne factorTwo

Class:	DFRational
Superclass:	DFAbstract
Instance variables:	numerator denominator

På vanligt sätt konstruerar vi sedan mutatorer och inspektorer för alla instansvariabler, men skriver inte ut dem. Dessa får som vanligt samma namn som respektive instansvariabel. Vi konstruerar också mutatorer med ett namn konstruerat från en kombination av instansvariablerna, tex i klassen DFProduct

```
factorOne: aValue1 factorTwo: aValue2
    self factorOne: aValue1.
    self factorTwo: aValue2
```

Slutligen skriver vi konstruktörer med namn från sammanslagna instansvariabelnamn, tex skriver vi följande metod i metaklassen DFSum class

```
termOne: aValue1 termTwo: aValue2
    ^self new termOne: aValue1 termTwo: aValue2
```

Vi skriver också några metoder som automatiskt ger oss lite snyggare utmatning vid utskrift av respektive instans. Principerna för ett objekts presentation gås igenom i kapitel 10.

DFConstant

printOn: aStream
"En konstant skrivs som enbart ett numeriskt värde, dvs på formen: x"
aStream nextPutAll: self number printString

DFMonom

printOn: aStream
"Ett monom skrivs på formen: fx^e, där f faktor och e exponent"
aStream nextPutAll: self factor printString; nextPutAll: self symbol asString;
nextPutAll: '^'; nextPutAll: self exponent printString

DFProduct

printOn: aStream
"En produkt skrivs på formen: (f1 * f2), dvs med parenteser för att förtydliga"
aStream nextPut: $(.
self factorOne printOn: aStream.
aStream nextPutAll: ' * '.
self factorTwo printOn: aStream.
aStream nextPut: $).

DFRational

printOn: aStream
"Skrivs på formen: (n / d)"
aStream nextPut: $(.
self numerator printOn: aStream.
aStream nextPutAll: ' / '.
self denominator printOn: aStream.
aStream nextPut: $).

DFSum

printOn: aStream
"Skrivs på formen: (t1 + t2)"
aStream nextPut: $(.
self termOne printOn: aStream.
aStream nextPutAll: ' + '.
self termTwo printOn: aStream.
aStream nextPut: $).

Hur det hela ter sig vid utmatning i ett textfönster framgår av exemplen som följer.

Aritmetik

Vi börjar med att skriva metoderna för att addera och multiplicera två givna objekt ur våra klasser. Då dessa metoder är starkt beroende av de objekt som ingår (dvs av mottagare och argument) indikerar vi endast deras nödvändighet i den abstrakta klassen DFAbstract. Dvs

```
+ anotherDF
    self subclassResponsibility
```

och

```
* anotherDF
    self subclassResponsibility
```

Vi sa att vi hade tänkt använda dubbel uppslagning. Detta medför att metoderna + och * i de konkreta subklasserna kommer att skicka ett nytt meddelande till argumentet (anotherDF) som beskriver hur mottagaren är beskaffad. De olika dispachmetoderna kommer se ut enligt följande:

```
+ anotherDF
    ^anotherDF sumFromKLASSNAMN: self
* anotherDF
    ^anotherDF productFromKLASSNAMN: self
```

Där ***KLASSNAMN*** ersätts suffixet till DF i respektive konkret klass namn, exempelvis metoden + i klassen DFRational ser ut som följer:

```
+ anotherDF
    ^anotherDF sumFromRational: self
```

Till att börja med kommer vi inte att utnyttja dubbel uppslagning utan endast använda generella metoder som vi implementerar i DFAbstract. Dessa metoder i ser alla ut som följer:

```
sumFromKLASSNAMN: aKLASSNAMN
    ^DFSum termOne: aKLASSNAMN termTwo: self
productFromKLASSNAMN: aKLASSNAMN
    ^DFProduct factorOne: aKLASSNAMN factorTwo: self
```

där ***KLASSNAMN*** återigen ersätts av varje konkret klassnamnssuffix, så att det finns en summerings och en multiplikationsuppslagning per klass. Tex ser metoden * i klassen DFMonom ut som följer:

```
productFromMonom: aMonom
    ^DFProduct factorOne: aMonom factorTwo: self
```

Anledningen till att vi gör oss besväret att använda dubbel uppslagning är att vi senare vill börja effektivisera koden och införa förkortningar på ett isolerat och strukturerat sätt men samtidigt ha möjlighet att testa koden även då vissa delar inte är ”färdiga”.

Innan vi skriver metoderna för subtraktion och division prövar vi koden som vi har skrivit hittills genom att addera och multiplicera ett par monom.

```
| m1 m2 |
m1 := DFMonom
    factor: (DFConstant number: 7)
```



```
        symbol: #x
        exponent: (DFConstant number: 5).
    m2 := DFMonom
        factor: (DFConstant number: 100)
        symbol: #x
        exponent: (DFConstant number: 2).
    m1 + m2
```

⟹*(7x^5 + 100x^2)*

Observera att vi måste ge konstanter som instanser av DFConstant, istället för att ge de numeriska värdena direkt. Anledningen är att om vi kan förlita oss alla objekt som bearbetas i de olika metoderna förstår meddelanden ur ”vår egen” objekthierarki så blir koden mycket elegantare än annars. Vi kan speciellt dra nytta av detta vid en senare förenkling av uttrycken.

Om vi istället multiplicerar de två monomen, dvs

```
m1 * m2
```

så blir resultatet

⟹*(7x^5 * 100x^2)*

Som framgår förenklas inga resultat, vilket i det här fallet hade inneburit att vi istället hade fått svaret 700x^7. Men som tidigare sagts förkortning är ett mer komplicerat problem som vi återkommer till senare.

Vi passar också på att definiera ett sista uttryck, för att även testa att det hela också fungerar för mer komplicerade uttryck. Här använder vi samma m1 och m2 som tidigare men definierar också en summa och en produkt och utför en addition och multiplikation enligt följande:

```
sum := DFSum termOne: (DFConstant number: 2) termTwo: m1.
p := DFProduct factorOne: (DFConstant number: 4) factorTwo: sum.
sum + p* m2
```

⟹*(((2 + 7x^5) + (4 * (2 + 7x^5))) * 100x^2)*

Nu kan vi slutligen, i det aritmetiska avsnittet, skriva metoderna för subtraktion och division. Dessa två metoder kan enkelt uttryckas på följande sätt i klassen DFAbstract:

```
- anotherDF
    ^self + anotherDF negated

/ anotherDF
    ^self * anotherDF reciprocal
```

Men vi har ju inte implementerat negated eller reciprocal, avsedda att precis som vanliga numeriska objekt returnera mottagaren negerad res-

pektive dess invers på följande sätt (med monomet m1 från tidigare som mottagare):

```
    m1 negated
⟹-7x^5
    m1 reciprocal
⟹(1/7)x^-5
```

Då dessa metoder är starkt beroende av den klass de tillhör, behöver varje klass implementera sin egen variant av dem båda.

```
DFConstant
  negated
      ^self species number: self number negated
  reciprocal
      ^self species number: self number reciprocal
DFMonom
  negated
      ^self species
          factor: self factor negated
          symbol: self symbol
          exponent: self exponent
  reciprocal
      ^self species
          factor: self factor reciprocal
          symbol: self symbol
          exponent: self exponent negated
DFProduct
  negated
      ^self factorOne negated * self factorTwo
  reciprocal
      ^self factorTwo reciprocal * self factorOne reciprocal
DFRational
  negated
      ^self numerator negated / self denominator
  reciprocal
      ^self denominator / self numerator
DFSum
  negated
      ^self termOne negated + self termTwo negated
  reciprocal
      ^DFConstant unity / self
```


Observera speciellt hur de olika metoderna förlitar sig på att objektets delar är ”av rätt typ” och alltså också förstår meddelandena negated och reciprocal.

Som vanligt kan den abstrakta klassen implementera dessa meddelanden, men förtydliga att respektive konkret klass måste konstruera sin egen metod med en subclassResponsibility.

Vi skriver också två konstruktörer i klassen DFConstant. Dessa gör det enkelt att införa vissa förbättringar utan att känna till alltför mycket om det inre av klassen

```
unity
    ^self number: Number unity

zero
    ^self number: Number zero
```

Vi skriver också följande kosmetiska metod:

```
DFAbstract

squared
    ^self * self
```

Derivering

När vi har skrivit de ovan beskrivna aritmetiska funktionerna så blir det väldigt enkelt att också införa metoder för symbolisk derivering.

Att derivera en konstant är enkelt, resultatet är ju alltid noll.

```
DFConstant

df: aSymbol
    ^self species zero
```

För en summa deriverar vi respektive term och bildar en ny summa av resultatet.

```
DFSum

df: aSymbol
    ^(self termOne df: aSymbol) +
        (self termTwo df: aSymbol)
```

En produkt följer också rakt fram från derivatans definition.

```
DFProduct

df: aSymbol
    ^(self factorOne df: aSymbol) * self factorTwo +
        (self factorOne * (self factorTwo df: aSymbol))
```

Ett rationellt tal är också rakt fram (det är här vi använder squared som vi definierade i DFAbstract).

DFRational

```
df: aSymbol
    ^(self numerator df: aSymbol) * self denominator -
        (self numerator * (self denominator df: aSymbol)) /
            self denominator squared
```

Vid derivering av ett monom råkar vi för första gången hittills på några problem. Vi måste först undersöka om symbolen, vi avser att derivera med avseende på, är densamma som monomets symbol, om inte ska resultatet bli noll, dvs DFConstant zero.

DFMonom

```
df: aSymbol
    self symbol ~= aSymbol
        ifTrue: [^DFConstant zero].
    ^self species
        factor: self factor * self exponent
        symbol: self symbol
        exponent: self exponent - DFConstant unity
```

Tester av deriveringsfunktionerna: 1

Triviala fallet:

```
(DFConstant number: 7) df: #x
```

⇒*0*

Monom:

```
m1 := DFMonom
    factor: (DFConstant number: 10)
    symbol: #x
    exponent: (DFConstant number: 5).
m1 df: #x.
```

⇒*(10 * 5)x^4*

```
m1 df: #y
```

⇒*0*

Mer komplicerat:

```
m2 := DFMonom
    factor: (DFConstant number: 3)
    symbol: #x
    exponent: (DFConstant number: 2).
(m1 + m1) * m1 / m2 df: #x
```

⇒*((((((10 * 5)x^4 + (10 * 5)x^4) * 10x^5) + ((10x^5 + 10x^5) * (10 * 5)x^4)) * (1/3)x^-2) + (((10x^5 + 10x^5) * 10x^5) * ((1/3) * -2)x^-3))*

Som vi ser har nu behovet av förkortningar avsevärt ökat. Det är inte helt uppenbart att resultatet av föregående exempel kan förenklas till:

$$\frac{1600}{3}x^7$$

Förkortning av resultat

Vi börjar med att lägga till några hjälpfunktioner, nämligen:

```
DFAbstract
  isUnity
      ^false
  isZero
      ^false
DFConstant
  = anotherObject
      ^(self isMemberOf: anotherObject)
          and: [self number = anotherObject number]
  isUnity
      ^self number = self class unity
  isZero
      ^self number = self class zero
DFProduct
  isUnity
      ^self factorOne isUnity and: [self factorTwo isUnity]
  isZero
      ^self factorOne isZero or: [self factorTwo isZero]
DFMonom
  isUnity
      ^self factor isUnity and: [self exponent isZero]
  isZero
      ^self factor isZero
DFSum
  isUnity
      ^(self termOne isUnity and: [self termTwo isZero])
      or: [self termTwo isUnity and: [self termOne isZero]]
  isZero
      ^self termOne isZero and: [self termTwo isZero]
```

Nu kan vi göra en liten ”optimering” i DFMonom>>df: och returnera noll om faktorn av någon anledning skulle vara noll eller enbart faktorn (self factor) om monomets exponent är ett (self exponent isUnity). Observera återigen att vi inte använder det numeriska värdet 1 (ett) utan det

objekt som vi har valt som enhetselement för våra egna klasser, dvs DFConstant unity.

```
df: aSymbol
    (self symbol ~= aSymbol or: [self factor isZero])
        ifTrue: [^DFConstant zero].
    self exponent isUnity ifTrue: [^self factor].
    ^self species
        factor: self factor * self exponent
        symbol: self symbol
        exponent: self exponent - DFConstant unity
```

Andra förenklingar lämnas till den intresserade läsaren.

Automatisk generering av inspektorer och mutatorer

Det är ganska vanligt att man vill konstruera klasser som har inspektorer och mutatorer för alla sina instansvariabler. Vidare brukar man ofta vilja att det ska finnas konstruktörer där man ges möjlighet att med ett enda meddelande ange initiala värden för alla instansvariabler.

För att understödja detta ska vi nu konstruera en metod som direkt, när klassdefinitionen görs i browsern, automatiskt konstruerar lämpliga åtkomstmetoder och konstruktörer.

För att det ska vara så enkelt som möjligt att ange att en klass ska genereras på detta sätt väljer vi att ge metoden samma namn som den vanliga klassdefinitionsmetoden, men med ett prefix record.

Rätta platsen för en metod av den här typen är i klassen Class.

Vi beskriver det hela uppifrån och ner och börjar med den ”publika” metoden.

```
recordsubclass: className instanceVariableNames:
instanceVariableNames classVariableNames: classVariableNames
poolDictionaries: pools category: category
    | newClass names |
    "Vi skapar först klassen på vanligt sätt"
    newClass := self
        subclass: className
        instanceVariableNames: instanceVariableNames
        classVariableNames: classVariableNames
        poolDictionaries: pools
        category: category.
    "Namnen på klassens alla instansvariabler"
    names := newClass instVarNames.
    "Vi skapar  inspektorer och mutatorer för alla instansvariabler"
    newClass createInspectorsAndMutatorsFor: names.
    "Vi skapar också en konstruktör med instansvariablerna som nyckelord"
    newClass createKeywordConstructorFor: names.
    ^newClass
```

Innan vi beskriver hjälpmetoderna så nämner vi att meddelandet

compile: aString classified: category

skapar en ny metod i mottagande klass där definitionen ges av aString och kategorin av category.

```
createInspectorsAndMutatorsFor: names
    "Skapar inspektorer och mutatorer för alla variabler givna i behållaren names"
    names
        do:
        [:aName |
            self createInspectorOf: aName.
            self createMutatorOf: aName]

createInspectorOf: namedVariable
    "Skapar en inspektor för argumentet"
    self compile: namedVariable , '
    ^' , namedVariable classified: #accessing

createMutatorOf: namedVariable
    "Skapar en mutator för argumentet"
    self compile: namedVariable , ': newValue
    ' , namedVariable , ' := newValue' classified: #accessing
```

Den avslutande hjälpmetodens beskrivning blir lite mer komplicerad då vi både konstruerar en nyckelordsmetod med automatiskt genererade namn på de formella parametrarna och kod för att anropa respektive mutator för att sätta respektive värde.

```
createKeywordConstructorFor: collectionOfVariableNames
    "Skapar  en konstruktör med ett namn bestående av alla olika
    instansvaribelnamn som nyckelord. Deras respektive initiala värden kan
    sedan ges som argument till metoden vid instansieringen.
    En instansmetod med samma namn skapas också"
    | methodName code |
    "Metodnamnet (methodName) och kodkroppen (code) börjar konstrueras"
    methodName := ''.
    code := String with: Character cr.
    1 to: collectionOfVariableNames size
        do: [:i |
            | thisName thisID |
            "thisName blir  nyckelord och thisID formell parameter"
            thisName := collectionOfVariableNames at: i.
            thisID := (thisName first isVowel
                        ifTrue: ['an']
                        ifFalse: ['a'])
                        , (thisName
                                changeFrom: 1 to: 1
                                with: (String with: thisName first asUppercase)),
                                                                i printString.
            methodName := methodName , thisName , ': ' , thisID , ' '.
            code := code , 'self ' , thisName , ': ' , thisID , '.
    '].
```

```
"Om antalet variabler är större än ett behöver vi konstruera en kombinerad
instansmetod. Vanliga mutatorer skall redan finnas."
collectionOfVariableNames size > 1
    ifTrue: [self compile: methodName , code classified: #accessing].
"Slutligen skapar vi konstruktörenn i kategorin instance creation"
self class compile: methodName , '
^self new ' , methodName classified: #'instance creation'
```

Vi prövar

Nu kan vi använda browsern och skriva in följande definition.

```
Object recordsubclass: #KlassAvRecordTyp
    instanceVariableNames: 'namn nummer telefon'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Record'
```

Om vi väljer menyalternativet **acccept** så utförs konstruktionen, där en kategori accessing skapas som innehåller mutatorer och inspektorer för alla instansvariabler samt följande instansmetod:

```
namn: aNamn1 nummer: aNummer2 telefon: aTelefon3
    self namn: aNamn1.
    self nummer: aNummer2.
    self telefon: aTelefon3.
```

Vidare konstrueras följande klassmetod i kategorin instance-creation:

```
namn: aNamn1 nummer: aNummer2 telefon: aTelefon3
    ^self new namn: aNamn1 nummer: aNummer2 telefon: aTelefon3
```

Sammanfattning

Termer

Metaklass en klass vars instanser är klasser.

Ersättare (eng. proxy) ett objekt uppträder som ett annat objekt. Ofta för att på ett transparent sätt vidarebefodra eller filtrera meddelanden till andra objekt.

Primitiv metod en metod som utförs direkt av den virtuella maskinen.

Smalltalk

Block är första ordningens objekt. Utförs när man skickar något value-meddelande till det. Är inblandade i och definierar vissa grundläggande styrstrukturer.

Boolean abstrakt klass med två konkreta subklasser True och False. är också första ordningens objekt. Beskriver villkorsates och logiska operationer.

UndefinedObject beskriver ickevärdet nil's beteende.

valueOnUnwindDo: samt valueNowOrOnUnwindDo: används för att hantera fel.

Alla variabler får initialt värdet nil om inget annat görs.

Alla komponenter är objekt, alla objekt tillhör klasser. Därför är också klasserna objekt som tillhör klasser.

Klasserna kan beskriva hur meddelanden som inte förstås av dess instanser ska hanteras. Detta görs i metoden doesNotUnderstand:.

Normalt är Object rot i klassträdet men det går att definiera nya träd med andra rötter. Vissa viktiga metoder som class och == går dock inte att definiera om utan blir ändå ekvivalenta med motsvarande metoder i Object.

Övningar

6.1 Skriv en klass för att hantera case-satser.

6.2 Skriv block som beräknar n!

a) Rekursivt

b) Iterativt

6.3 Skriv ett block som vid anrop avgör om argumentet är ett primtal eller ej.

6.4 Skriv ett kodavsnitt som kopierar alla metoder i en klass till en annan. Tänk på att också kopiera klassmetoder.

6.5 Skriv en ny typ av ersättare som registrerar hur lång tid alla metodanrop till objektet tar. Tips: använd Time millisecondsToRun:[...].

6.6 Skriv ett kodavsnitt som returnerar en klass källkod i form av en sträng. Tänk på att få med klassmetoderna.

6.7 Skriv en variant på metoden recordsubclass:... beskriven på sidan 238 som också ger en möjlighet att ange skönsvärden för instansvariablerna till exempel på följande sätt

```
Object initializedsubclass: #Person
    instanceVariableNames: #(name '''no name'''
                             age '48')
    classVariableNames:''
    poolDictionaries: ''
    category: 'Bok övningar'
```

där andra argumentet är en vektor (Array) där vart annat element är instansvariabelns namn och vart annat är en sträng som beskriver initieringen av resp instansvariabel. I det här fallet kan man tänka sig att följande metod genereras automatiskt:

```
initialize
    "initiering av nya instanser"
    super initialize.      se till att eventuell initialize metod i superklassen anropas
    name := 'no name'.
    age := 48.
```

6.8 Skriv metoder för att förenkla deriveringsuttrycken.

7 Behållarklasser

Detta kapitel besvarar bland annat följande frågor:

- Vad är en behållarklass?
- Vad finns det för olika typer av behållare?
- Vilka meddelanden kan alla behållare förstå och vad gör de?
- Vad bör jag välja för behållare i min tillämpning?
- Hur kan man skapa nya behållare?
- Hur ska jag göra om ingen behållare passar direkt?

I det här kapitlet ska vi studera klassen Collection som är en mycket viktig del av Smalltalk. Collection och dess subklasser är en klasshierarki som hanterar objekt som fungerar som behållare för andra objekt. Vi kallar objekten som finns i behållarobjekten för element. Collection-hierarkin hanterar allt som består av flera instanser av andra objekt tex vektorer, stackar, listor, paletter, strängar och texter. För att kunna arbeta effektivt med Smalltalk är det viktigt att känna till de olika klasserna, veta vad som skiljer dem åt och när en klass är att föredra framför en annan. Framför allt är det viktigt att veta vad som förenar dem. Principerna som visas i detta kapitel gäller förstås alla objektorienterade språk.

7.1 Bakgrund

Om alla programmerare skulle konstruera sina egna behållarklasser så skulle det leda till kompatibilitetsproblem mellan olika implementatörer. Lyckligtvis har man i Smalltalk redan från början insett att dessa klasser bör höra till standarduppsättningen, inte minst som Smalltalksystemet själv utnyttjar instanser av sådana klasser för att lösa sina grundläggande tjänster. Tex i fönsterhantering, beroenden mellan objekt, metodkataloger, kategorisering, stränghantering, paletter och mycket, mycket mer.

Systemet har redan från början en stor mängd klasser av sådan dynamisk typ. I så gott som alla tillämpningar klarar man sig med de fördefinierade klasserna och behöver ytterst sällan konstruera egna behållarklasser. När man behöver ett objekt med dynamisk struktur så ska man först och främst fundera över om någon av de befintliga klasserna kan utnyttjas och endast som en sista utväg implementera en egen dynamisk klass och då i första hand som aggregat och i andra hand som subklass till någon av de befintliga Collection-klasserna.

Olika typer av behållarklasser

Vi kan skilja mellan de klasser där elementen har en inbördes ordning och de där de är oordnade. Vi kan också skilja mellan klasser där elementen går att komma åt mha index och klasser där detta inte går. Denna uppdelning ger oss fyra huvudtyper av behållare. Nedanstående figur visar dessa och exempel på minst en viktig klass ur varje grupp.

Behållartyper	Sekventiell ordning	Utan ordning
indexerbara	Array & String	Dictionary
ej indexerbara	LinkedList	Set

Figur 7.1 Huvudtyper av behållare och exempel på klasser

7.2 Behållarklassernas rot

Den viktigaste klassen att känna till och förstå är Collection men den kommer du aldrig att använda! Collection är en abstrakt klass som beskriver det gemensamma gränssnittet för sina subklasser. Eftersom Collection är en abstrakt klass så är inte alla metoder implementerade, vissa metoder säger bara att det är upp till subklassen att implementera dem, men vi får veta att de ska finnas och ska vara implementerade i dess subklasser. I dessa metoder i Collection är den enda raden i metodkroppen:

```
self subclassResponsibility
```

Anropar man en sådan metod kommer det att resultera i ett felavbrott. Eftersom det finns många olika typer av mängdklasser, avsedda för vitt skilda ändamål, så är det inte säkert att alla metoder i Collection är meningsfulla utan de kan vara ”borttagna” i subklasserna. Detta ser vi genom att metodkroppen i subklassen är:

```
self shouldNotImplement.
```

Vilket också resulterar i ett felavbrott om man försöker anropa den. Vad ger då Collection för möjlighet att hantera element? Det finns i huvudsak fem olika saker man kan göra:

- lägga till objekt
- ta bort objekt
- indexerad åtkomst
- göra något med varje objekt
- ändra typ av behållare

Innan vi visar hur man gör detta ska vi titta på hur man skapar en behållare.

Skapa behållare

Det finns i huvudsak tre olika sätt att skapa en behållare. Antingen definierar man behållaren med innehåll från dess litterala form, genom att konvertera från en behållartyp till en annan eller genom ett meddelande till den klass ur vilken man vill skapa behållare.

Den litterala konstruktionsformen bygger på att Smalltalks kompilator känner igen ett visst sätt att beskriva en behållare och dess innehåll. Ofta ger det här sättet att konstruera behållare den kortaste och mest lättlästa koden. Exempel:

```
enOrdVektor := #('ett' 'två' 'tre').
```

Att konvertera en behållare från en typ till en annan är ofta praktiskt om man redan har en behållare och endast typen behöver ändras. Metoder för att konvertera objekt brukar börja med 'as', följt av den klass som ska skapas t ex asSet. Exempel:

```
enOrdMängd := enOrdVektor asSet.
```

Den tredje möjligheten är att skicka meddelanden direkt till den klass som man vill skapa en instans av. Som vi redan har sett så kan vi skapa behållare med new:, där argumentet avgör behållarens storlek. Vi kan också direkt lägga in objekt i behållaren genom att skicka with-meddelanden. Det finns fem olika:

with: ettElement	Skapar ny behållare och lägger till ettElement som enda element
with: element1 with: element2	Skapar ny behållare med två element; element1 och element2
with: with: with:	Skapar ny behållare med tre element (element1-3 utelämnade)
with: with: with: with:	Skapar ny behållare med fyra element (element1-4 utelämnade)
withAll: aCollection	Skapar en ny behållare där alla element i aCollection läggs in

Som exempel på hur behållare kan skapas utgår vi från klasserna Array och Set. Vi börjar med att konstruera en Array och använder då den litterala formen:

```
vector := #('en' 'kort' 'array').
```

Ett alternativt sätt att skapa en vektor med elementen 'en', 'kort' och 'array', är med with-meddelande:

```
vector := Array with: 'en' with: 'kort' with: 'array'
```

Vill vi ha en mängd med samma element kan vi konvertera vektorn:

```
aSet := vector asSet.
```

Variabeln aSet kommer nu vara av typen Set och innehålla samma element som vector. Efter satsen så är variabeln vector helt opåverkad.

Lägga till element

För att lägga till element använder man någon av metoderna under kategorin *adding*. Dessa meddelanden går att använda för de flesta typer av behållare, dock inte för Array och Dictionary-klasser där man även måste ange var de ska stoppas in. De viktigaste add-metoderna är:

add: ettElement	Som lägger till ettElement till behållaren. Om behållaren är ordnad så hamnar elementet sist
addAll: aCollection	Lägger till alla element i aCollection till behållaren med hjälp av add:

Om behållaren är ordnad så finns också metoderna

addFirst: ettElement	Lägger till ettElement först
addLast: ettElement	Lägger till ettElement sist
add: enElement before: annatElement	Lägger till ettElement före annatElement

När ett objekt adderas med någon av dessa metoder så kommer objektet också att returneras som resultat från metoden. Alltså inte mottagaren av meddelandet, dvs mängdobjektet, som man skulle kunna tro. Vad får det för effekt för följande sats?

```
newList := OrderedCollection new add: 1
```

Först skapas en OrderedCollection, därefter läggs '1' till listan och därefter tilldelas variabeln newList '1'! För att få newList att referera till en lista kan vi göra följande:

```
newList := OrderedCollection new.
newList add: 1.
```

eller med kaskadmeddelande:

```
newList := OrderedCollection new add: 1 ; yourself
```

Ta bort objekt

Det är möjligt att ta bort objekt på motsvarande sätt som man lägger till objekt. För att hitta sådana metoder söker man under kategorin *removing*. Vi kan ta bort ett visst objekt med remove: anObject. Andra metoder för att ta bort objekt är:

remove: ettObject	tar bort argumentet, ettObject, ur mottagaren
removeAll: enSamlingObjekt	tar bort alla objekt i enSamlingObjekt ur mottagaren

Om vi har en behållare där elementen är ordnade så kan vi också använda metoderna:

removeFirst	Tar bort första elementet i behållaren
removeFirst: antalElement	Tar bort de första antalElement ur behållaren
removeLast	Tar bort sista elementet i behållaren
removeLast: antalElement	Tar bort de sista antalElement ur behållaren

Det finns en del andra metoder för att ta bort element ur behållare definierade i de olika subklasserna. Titta i browsern för att se vilka!

Precis som add-operationerna så returnerar remove-operationerna argumentet och inte behållarobjektet.

Indexerad åtkomst

För vissa typer av behållare kan man komma åt och ändra innehållet med hjälp av index. Exempelvis använder Array heltalsindex.

```
| vektor |
vektor := Array new: 2.
vektor at: 1 put: 'Smalltalk är ett rent objektorienterat språk'.
vektor at: 2 put: #'symboler kan innehålla mellanslag om de omges av apostrofer'
```

Här konstruerar vi först en Array med storleken två. För att lägga in objekt på en viss position används at:put:. Först lägger vi in strängen 'Smalltalk är ett rent objektorienterat språk' i position ett. Därefter lägger vi in en symbol i position två. När vi har gjort det är vektorn full. Nu kan vi komma åt objekten med at:

```
symbolen := vektor at: 2.
strängen := vektor at: 1.
```

Index behöver inte alltid vara heltal utan kan för vissa typer av behållare vara godtyckliga objekt.

Göra något för varje element

Eftersom det finns många olika typer av behållare med olika gränssnitt och datastruktur så är det viktigt att ändå ge användaren ett enhetligt gränssnitt. Blant annat kan alla element i en behållare gås igenom med en *iterator*. En gemensam egenskap hos dessa iterationsmetoder är att de ska ges med ett block som argument. Blocket kommer för de flesta iteratorer att utföras en gång för varje element i behållaren. Iteratorerna kan man använda både för att förändra objekten i behållaren och för att välja ut ett eller flera objekt ur behållaren.

Nedan följer en lista med korta förklaringar till de olika iteratorerna:

do: ettBlock	Utför ettBlock på varje element i behållaren.
collect: ettBlock	Ger en ny behållare av samma typ som den gamla med ettBlock utfört på alla element.
select: ettBlock	Returnerar en behållare med alla element som returnerar true då de skickas som argument till ettBlock.
reject: ettBlock	På samma sätt som select:, men elementen som returnerar false
detect: ettBlock	Returnerar det första objekt för vilket ettBlock blir true. Om inget objekt ger true genereras ett felavbrott.
detect: ettBlock ifNone: omIngetBlock	Returnerar det första elementet för vilket ettBlock blir true. Om inget true-element hittas så utförs omIngetBlock

do: ettBlock separatedBy: ettAndraBlock	Utför ettBlock för varje element, precis som do:, men dessutom utförs ettAndraBlock mellan varje gång ettBlock anropas
inject: startVärde into: ettBinärtBlock	Första gången utförs ettBinärtBlock med startVärde och första element som argument. Andra gången utför ettBinärtBlock med resultatet av första anropet av blocket och andra elementet osv. Som resultat av meddelandet returneras sista anropet till ettBinärtBlock

Om det finns en ordningsrelation i behållaren så kan man använda även följande metoder:

reverseDo: ettBlock	Utför ettBlock för varje element, men i omvänd ordning mot do:
findFirst: ettBlock	Returnerar index för första element för vilket ettBlock ger true
findLast: ettBlock	Returnerar index för sista element för vilket ettBlock ger true
keysAndValuesDo: ettBinärtBlock	Utför ettBinärtBlock för varje element. Argumenten till ettBinärtBlock är elementets nyckel och elementet själv
reverse	Returnera en ny behållare med elementen i omvänd ordning
with: enOrdnadLista do: ettBinärtBlock	Itererar över två listor samtidigt, mottagaren av meddelandet och enOrdnadLista. För varje element ur de båda listorna anropas ettBinärtBlock. Listorna måste vara lika långa

För de behållare som har element med en inbördes ordning finns det meddelanden som skapar en ny ordnad behållare

, enBehållare	Lägger argumentets element till mottagarens i en ny behållare. Används ofta för att slå ihop teckensträngar
copyFrom: start to: stopp	Skapar en ny behållare som består av alla element hos mottagaren from index start tom stopp.

copyWith: nyttElement	Skapar en ny behållare som består av motagaren med nyttElement adderat sist. Kan i vissa fall användas istället för add:.
copyWithout: gammaltElement	Skapar en ny behållare med alla förekomster av gammaltElement borttagna.
copyReplaceAll: bytUt with: mot	Skapar en ny behållare med alla förekomster av behållaren bytUt ersatta med behållaren mot's element. Används typiskt på strängar får att ersätta alla förekomster av ett ord mot ett annat

Ändra typ av behållare

Eftersom det finns olika typer av behållare så händer det att man behöver konvertera en behållare till en annan typ. Konverteringar görs genom att skicka unära meddelanden som talar om hur man vill konvertera behållaren. Exempel på sådana meddelanden är asSet, asOrderedCollection och asSorterdCollection. Egentligen konverteras inte behållaren utan en ny behållare skapas med elementen i den ursprungliga behållaren instoppade.

7.3 Collections klasshierarki

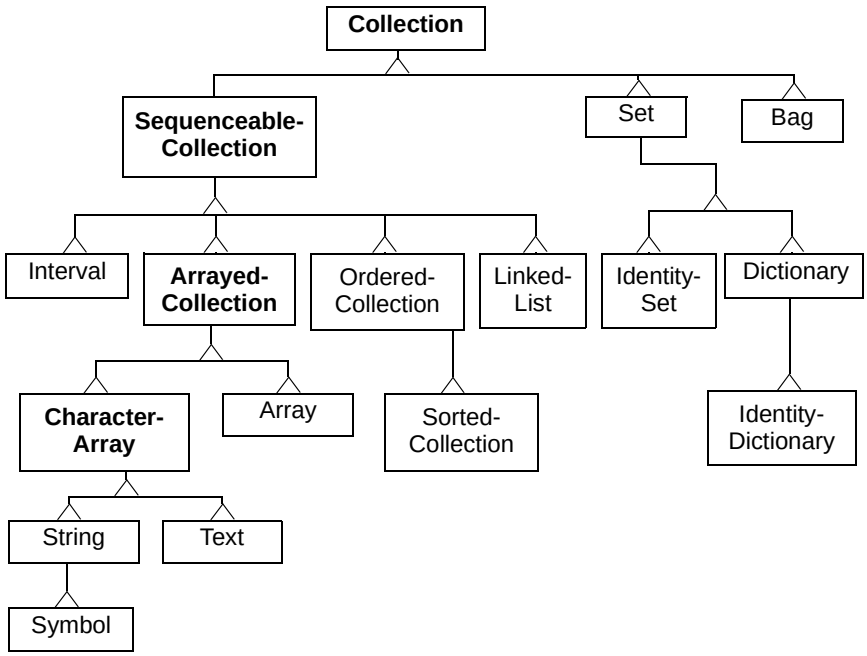
Smalltalksystemets behållarklasser är uppbyggda med en klasshierarki. Hierarkin för de viktigaste behållarna finns i figur 7.2:.

Hierarkin delas upp mellan ordnade och oordnade behållare, där alla ordnade är subklasser till SequencableCollection.

Egenskapen om en klass är indexerbar är lite svårare att se, eftersom det inte finns någon arvshieraki med indexerbara respektive icke indexerbara behållare. För att avgöra om en behållare är indexerbar måste vi titta på definitionen av metoden at: och at:put: och se om dessa är definierade eller ej. Klasserna Bag, Set och IdentitySet är inte alls indexerbara medan klasserna LinkedList och Interval bara förstår at: och inte at:put: och blir ett mellanting, indexerbara vid inspektion men inte vid insättning.

Konkreta subklasser till Collection

I de följande avsnitten kommer vi att väldigt kort beskriva några av de viktigaste subklasserna till Collection. Mer avancerad användning kommer i efterföljande exempel och övningar. Vidare sparar vi beskriv-



Figur 7.2 Hierarki för de viktigaste behållarklasserna.
Klassnamn i fetstil är abstrakta klasser, övriga konkreta

ningen av hur de abstrakta mängdklasserna används till en diskussion om utnyttjande av arvsmekanismer i kapitel 8.

7.4 Array

En Array tillåter att en positionering av de ingående elementen sker. Ett elements position avgörs av heltalsindex.

Form

Heltalsindexering, snabb åtkomst av element.

Konstruktion

Nya konstrueras med new:, element adderas med at:put:.

```
| arr |
arr := Array new: 3.
arr at: 1 put: 1.
arr at: 3 put: Time now.
arr
```

⇒*#(1 nil 8:46:26 am)'*

Litteral form

En instans av Array kan också konstrueras med en litteral beskrivning #(…). Inom parentesen kan man skriva in tal, strängar, symboler, true, false, nil och vektorer. Exempel :

```
anArray := #(12 25.4 3/4 'en sträng' enSymbol true false nil)
anArray
```

⇒*#(12 25.4 3/4 'en sträng' #enSymbol true false nil)*

där de första tre elementen är tal, sedan följer en sträng, en symbol och till sist objekten true, false och nil.

Konvertering

En instans av Array förstår en stor uppsättning av meddelanden för att konstruera en kopia av den aktuella vektorn som en instans av någon av de övriga grundläggande Collection-klasserna. Exempel finns i beskrivningarna av de övriga Collection-klasserna nedan. Av symmetriskäl kan en vektor också konverteras till en vektor.

```
#(10 5 7 1 8 2 5 7) asArray
```

⇒*#(10 5 7 1 8 2 5 7)*

Övrigt

Storleken av en Array anges då den skapas och kan sedan inte påvekas.

7.5 Bag

Klassen Bag är den enklaste av de olika mängdhanterande klasserna. Elementen är oordnade och at: kan inte användas. Till en instans av Bag kan nya objekt adderas med add: och gamla objekt tas bort med remove: För att undersöka om ett visst objekt finns i en Bag kan includes: användas. En Bag håller reda på villka objekt som finns i den och även antalet dubbletter.

Form

Behållare för oordnad lagring av objekt. Ingen användnig av index, ingen sortering, räknar dubbletter.

Konstruktion

Nya konstrueras med new, element adderas med add:.

```
| collection |
collection := Bag new.
collection add: 1; add: 7; add: 3; add: 7; add: 'a string'; add: 4.
collection
```

⇒*Bag (1 3 4 'a string' 7 7)*

Litteral form

Litteral form saknas. Det är möjligt att utgå från en Array och sedan konvertera till Bag.

```
#(10 5 7 1 8 2 5 7) asBag
```

⇒*Bag (1 2 5 5 7 7 8 10)*

Konvertering

För att konvertera en behållare till en Bag så används asBag.

Övrigt

Då ett objekt lagras i en Bag så kontrolleras först om objektet redan finns där, om det inte gör det så läggs det till. Om det redan finns så kommer istället en räknare att räknas upp. På motsvarande sätt räknas räknaren ner då element tas bort. Finns det endast ett element kvar så tas det bort. Genom att en räknare är kopplad till varje element så sparas minnesutrymme när man har dubbletter.

7.6 Set

Klassen Set beskriver det matematiska begreppet mängd. Elementen i en instans av Set är oordnade och har inga externa nycklar. Ett Set är i stort sett som en Bag förutom att dubbletter inte räknas. Om man försöker att lägga till ett element till ett Set så förändras mängden endast om elementet inte tidigare ingick i den.

Form

Inga index, ingen sortering, ignorera dubbletter.

Konstruktion

Nya konstrueras med new, element adderas med add:.

```
| collection |
collection := Set new.
collection add: 1; add: 7; add: 3; add: 7; add: 'a string'; add: 4.
collection
```

⇒*Set (7 1 3 4 'a string')*

Till skillnad mot en Bag ovan så kommer siffran 7 bara med en gång.

Litteral form

Litteral form saknas men det går att konvertera en vektor till ett Set med det unära meddelandet asSet.

```
#(10 5 7 1 8 2 5 7) asSet
```

⇒*Set (1 2 5 7 8 10)*

Konvertering

Konvertering sker med asSet.

Övrigt

Den interna represenationen av Set bygger på en hashtabell och att varje objekt kan returnera ett hashvärde. Det betyder att det går väldigt fort att avgöra om ett visst objekt finns i ett Set.

Vi avslutar denna beskrivning av Set med att säga några ord om hur instanser avgör om två objekt är lika eller inte. Det finns i huvudsak två typer av likhet nämligen referenslikhet, som normalt testas med ==, och värdemässig likhet, testas normalt med =. Likhet mellan element i ett Set är definierad med = och det är detta som används för att kontrollera om ett element redan finns i mängden. Exempel:

```
| x y s |
x := #(1 2 3) asOrderedCollection.
y := #(1 2 3) asOrderedCollection.
s := Set new.
s add: x; add: y.
s
```

⇒*Set (OrderedCollection (1 2 3))*

Som användare av Set måste man vara medveten om detta. Betrakta tex följande utvidgning av det föregående exemplet:

```
| x y s |
x := #(1 2 3) asOrderedCollection.
y := #(1 2 3) asOrderedCollection.
s := Set new.
s add: x; add: y.
x add: $x.
y add: $y.
```

Nu ser s ut som följer:

```
Set (OrderedCollection (1 2 3 $x ) )
```

trots att x och y är ej längre lika då y nu ser ut så här:

```
OrderedCollection (1 2 3 $y )
```

Lösningen på detta problem, om vi vill att x och y ska ha full integritet, är ett IdentitySet.

Om elementen i ett Set har definierat om = så ska de också definiera om metoden hash för att uppföra sig på ett förväntat sätt.

7.7 IdentitySet

Implementation av det matematiska begreppet Set. För att avgöra objektlikhet så används referenslikhet ==. Det betyder att skillnaden mellan IdentitySet och Set är att IdentitySet använder == istället för = vid jämförelse mellan ingående element.

Form

Mängd med referenslikhet.

Konstruktion

För att beskriva skillnaden mellan de båda mängdklasserna så tittar vi på exemplet från beskrivningen av Set än en gång men nu använder vi IdentitySet istället.

```
| x y s |
x := #(1 2 3) asOrderedCollection.
y := #(1 2 3) asOrderedCollection.
s := IdentitySet new.
s add: x; add: y.
x add: $x.
y add: $y
s
```

⇒*IdentitySet (OrderedCollection (1 2 3 $x) OrderedCollection (1 2 3 $y))*

Konvertering

Det finns ingen unär metod för att konvertera en behållare till IdentitySet. Däremot så kan klassmetoden withAll: användas för att enkelt skapa ett nytt IdentitySet.

```
| array identSet |

array :=#('hej'  7 1 'hej'  7 3).
identSet := IdentitySet withAll: array.
identitySet
```

⇒*IdentitySet(7 'hej' 1 3 'hej')*

Det framgår att de två strängarna 'hej' inte är samma objekt medan däremot de två siffrorna 7 i array är det. Strängar är sammansatta objekt medan tal är litterala konstanter.

Övrigt

Ännu snabbare åtkomst än Set.

7.8 OrderedCollection

En OrderedCollection är som en Array, men tillåter att element på ett friare sätt läggs till eller tas bort. En OrderedCollection definierar också en ordning mellan elementen.

En OrderedCollection använder ett heltalsindex och accepterar godtyckliga objekt som element. Den förstår också meddelanden som add: och remove:.

Form

Heltalsindex, dynamisk tillväxt.

Konstruktion

Nya konstrueras med new eller effektivare med new: om man vet storleken. Element adderas med add:.

```
| collection |
collection := OrderedCollection new.
collection add: 1; add: 7; add: 3; add: 7; add: 'a string'; add: 4.
collection
```

⇒*OrderedCollection (1 7 3 7 'a string' 4)*

Vi kan fortsätta exemplet med att ändra några av objekten i collection:

```
collection  at: 1 put: 'hej'.
collection
```

⇒*OrderedCollection ('hej' 7 3 7 'a string' 4)*

Litteral form

Litteral form saknas men vi kan gå via Array.

```
#(10 5 7 1 8 2 5 7) asOrderedCollection
```

⇒*OrderedCollection (10 5 7 1 8 2 5 7)*

Konvertering

Konverteringsmetoden är asOrderedCollection. Den finns definierad i Collection så alla behållare förstår den.

Övrigt

Klassen OrderedCollection är användbar för att hantera objekt som är inbördes ordnade. Klassen har ett rikt gränssnitt så att objekt kan läggas till och tas bort på flera olika sätt. Titta gärna i kategorierna adding och removing. Man bör tänka på att en OrderedCollection bygger på en vektor och därför inte är effektiviserad för att lägga till och ta bort element annat än i början eller slutet, även om det går.

7.9 SortedCollection

Klassen SortedCollection upprätthåller en ordning mellan elementen. Om man skapar en SortedCollection och slumpvis lägger till strängar så kommer de sorteras in i ordning efter den teckenkod som används.

En SortedCollection sorterar element som adderas efter given sorteringsalgoritm, det så kallade sorteringsblocket. Sorteringsblocket kan definieras av användaren. Skönsvärdet för blocket är: [:x :y | x <= y].

Form

Sortering, heltalsindex, dynamisk tillväxt.

Konstruktion

Först ett enkelt exempel där talen sorteras i storleksordning:

```
| collection |
collection := SortedCollection new.
collection add: 1; add: 7; add: 3; add: 7; add: 4.
collection
```

⟹*SortedCollection (1 3 4 7 7)*

Om vi istället vill ha talen sorterade fallande så skriver vi:

```
| collection |
collection := SortedCollection new.
collection sortBlock: [:x :y | x > y].
collection add: 1; add: 7; add: 3; add: 7; add: 4.
collection
```

⟹*SortedCollection (7 7 4 3 1)*

Litteral form

Någon litteral form finns inte heller för SorteredCollection utan vi får gå den vanliga vägen via Array.

```
sc := #(10 5 7 1 8 2 5 7) asSortedCollection.
```

Konvertering

För denna klass finns det två olika konverteringsmetoder. Dels asSortedCollection som använder standardblocket vid sortering och dels asSortedCollection: aBlock där ett sorteringsblock ges som argument. Blocket lagras i instansen.

```
#(10 5 7 1 8 2 5 7) asSortedCollection
```
⇒*SortedCollection (1 2 5 5 7 7 8 10)*
```
sc := #(10 5 7 1 8 2 5 7) asSortedCollection: [:x :y | x > y]
```
⇒*SortedCollection (10 8 7 7 5 5 2 1)*
```
sc add: 17.
```
⇒*SortedCollection (17 10 8 7 7 5 5 2 1)*

Övrigt

SortedCollection använder insättningssortering för att lägga till enstaka objekt. Läggs många objekt till på samma gång så kommer QuickSort att användas.

7.10 Dictionary

En instans av Dictionary lagrar elementen med ett objekt som extern nyckel och ett annat objekt som värdet vid denna nyckel. Det är alltså normalt den externa nyckeln man använder för att hitta värdet. Som nyckel i ett Dictionary kan man använda vilken typ av objekt som helst. Man kan se Array som en form av Dictionary där nyckeln måste vara ett heltal inom en storleksgräns.

Form

Använder nyckel-värde-par för en katalogliknande uppslagning.

Konstruktion

Nya konstrueras med new eller effektivare med new: om man vet storleken. Element adderas med at:put:.

I följande exempel ska vi göra en engelsk-svensk ordbok bestående av endast tre ord med hjälp av ett Dictionary.

```
| dict |
dict := Dictionary new.
dict at: 'one' put: 'ett'.
dict at: 'two' put: 'två'.
dict at: 'three' put: 'tre'.

dict at: 'two'
```
⇒*'två'*

Vi kan också definiera vad som ska hända om den angivna nyckeln inte finns. Om vi i det ovan definierade registret frågar efter värdet för

objektet vid nyckeln 'x' så får vi ett felavbrott. Vi kan förhindra det genom att ange att strängen 'Not present' ska returneras om ingen nyckel hittas.

```
dict at: 'x' ifAbsent: ['Not present']
```

⟹*'Not present'*

I exemplet har vi använt strängar som både nycklar och värden, men ingenting hindrar oss från att använda objekt av godtycklig typ för både nycklar och värden.

Litteral form

Någon litteral form finns inte och det är inte heller så rättframt som för de tidigare klasserna att initiera ett Dictionary. För att ändå slapa ett Dictionary från en litteral beskrivning kan vi först skapa en Array.

```
#(#nyckel1 'värde1' 'nyckel2' #värde2 'nyckel3' #osv)
```

Dvs nyckel och värde omväxlande. För att från denna vektor skapa ett Dictionary så skickar vi meddelandet withKeysAndValues: till Dictionary.

```
|a|
a := #(#nyckel1 'värde1' 'nyckel2' #värde2 'nyckel3' #osv)
Dictionary withKeysAndValues: a
```

⟹*Dictionary(#nyckel1-> 'värde1' 'nyckel2' ->#värde2 'nyckel3' ->#osv)*

Konvertering

Det är svårt att hitta entydiga konverteringar från Dictionary till andra objekt eftersom ett dictionary består av en nyckel–värde koppling vilket ingen annan klass gör. Därför måste man specificera om nycklarna, värdena eller båda avses. Följande metoder kan användas:

- values
  OrderedCollection med alla värden
- keys
  Set med alla nycklar
- associations
  OrderedCollection med nycklar och värden som instanser av Association

Övrigt

Dictionary använder ett Set för den interna representationen. Det gör att uppslagningen går mycket snabbt.

7.11 IdentityDictionary

Klassen IdentityDictionary fungerar på samma sätt som Dictionary med den skillnaden att nycklarna jämförs med referenslikhet == istället för värdelikhet. Dictionary och IdentityDictionary förhåller sig till varandra på samma sätt som Set och IdentitySet gör

Form

Använder nyckel-värde-par för en katalogliknande uppslagning. Snabbare än Dictionary.

Konstruktion

Om vi skapar ett IndentityDictionary med strängar som nyckar får det följande effekt:

```
| dict |
dict := IdentityDictionary new.
dict at: 'one' put: 'ett'.
dict at: 'two' put: 'två'.
dict at: 'three' put: 'tre'.

dict at: 'two' ifAbsent: [‘nyckel finns inte’]
```

⇒*‘nyckel finns inte’*

Orsaken är att de två strängarna 'two' inte är samma objekt, även om de innehåller samma tecken. Vi försökte alltså slå upp ett objekt som inte fanns som nyckel i dict. Om vi däremot använder symboler som nycklar så fungerar vår kod som önskat.

```
| dict |
dict := IdentityDictionary new.
dict at: #one put: 'ett'.
dict at: #two put: 'två'.
dict at: #three put: 'tre'.

dict at: #two ifAbsent: [‘nyckel finns inte’]
```

⇒*‘två’*

Skälet till detta är att symboler är unika objekt och endast en symbol med ett visst namn existerar.

Konvertering

Se Dictionary ovan.

Övrigt

Snabbare än Dictionary, men kräver unika objekt som nycklar.

7.12 String

En objekt ur String eller någon av dess subklasser beskriver beteendet hos teckensträngar.

Form

En vektor av tecken som som beskriver en teckensträng.

Litteral form

Den litterala formen för att konstruera en sträng har vi redan använt många gånger och den är:

```
'detta är en sträng'
```

dvs en teckensträng omsluten av apostrofer.

Om man vill att en apostrof ska ingå i strängen så skriver man två stycken apostrofer direkt efter varandra:

```
s := 'Ordet ''hej'' är omgivet av apostrofer'.
Transcript show: s.
```

⇒*Ordet 'hej' är omgivet av apostrofer*

Konvertering

För att typkonvertera ett objekt till en sträng skickas asString till objektet. Endast klasser som på ett naturligt sätt kan ses som en sträng kan konverteras till sträng, tex symboler. Meddelandet printString förstås av alla objekt och returnerar en textsträng som beskriver objektet.

För att konvertera en sträng till versaler används asUppercase. Motsvarande asLowercase konverterar till gemena.

Konstruktion

För att slå ihop strängar finns det binära meddelandet ',' (komma) som skapar en ny sträng som bestående av den första strängen direkt följd med den andra. Detta kan användas på följande sätt:

```
vector := #(1 3 5 7 9).
outString := 'vektorns första element: ' , (vector at: 1) printString
        , ' och sista: ' , (vector at: vector size) printString.
Transcript show: outString
```

⇒*vektorns första element: 1 och sista: 9*

Här slår vi ihop alla delsträngar och tilldelar resultatet till variabeln outString och skriver sedan ut den i utskriftsfönstret.

Normalt skapas strängar från deras litterala form, men ibland är det bättre att utgå från klassmetoden with: för att tex skapa en sträng med ett specialtecken i. Om vi vill skapa en sträng bestående av endast ett tabulatortecken, så sker det tydligast med:

```
str := String with: Character tab
```

istället för:

```
str := ''
```

Som också är en sträng med endast ett tabtecken i. Som synes så syns det inte!

Övrigt

I String finns det också metoder för att leta efter en viss delsträng, plocka ut delsträng ur en längre sträng. Dessa metoder finns det flera olika varianter på. Botanisera i String, det lönar sig! String har också subklasser som beskriver strängar som har tecken som inte ingår i ISO-standarden 6937 tex tecknen å,ä och ö. Det betyder att om man skapar en sträng genom att i den litterala formen skriva å, ä eller ö så skapas en speciell sträng som kan hålla dessa tecken. Det här är normalt ingenting som man behöver tänka på utan konverteringen sker helt transparent. Detta är ett skäl till varför man bör testa på objekts typ med isString och inte med class.

7.13 Symbol

För att beskriva metodnamn används objekt ur Symbol eller någon av dess subklasser. Den viktigaste egenskapen hos symboler är att två symboler med samma namn också är samma objekt dvs

```
#enSymbol == #enSymbol
```

⇒*true*

Till skillnad från tex

```
'enSträng' == 'enSträng'
```
⇒*false*

Det betyder bla att det går väldigt snabbt att avgöra om två symboler är lika, till skillnad mot tex strängar där man måste jämfora tecken för tecken.

Form

En vektor av tecken som beskriver symbolen.

Litteral form

Den litterala formen för att konstruera en symbol är:

```
#enSymol
```

Det går också att skapa symboler som innehåller blanka genom att skriva symbolens sträng innom apostrofer:

```
#'detta är också en symbol'
```

Konvertering

För att konvertera objekt till symboler används asSymbol som är definierad i CharacterArray. Detta medför att alla CharacterArray's subklasser bla String, Text och Symbol förstår asSymbol.

Övrigt

För att se till att det bara finns en symbol med ett visst namn så håller klassen Symbol reda på alla instanser. När man konstruerar en symbol så kontrollerar klassen först om det redan finns en sådan symbol och i så fall returneras det objektet. Är det en ny symbol så skapas symbolen och lagras i klassen.

7.14 Text

Text beskriver precis som String en lista av tecken , men innehåller dessutom information om teckensnitt, storlek, typsort och liknande.

Med Text kan man beskriva i detalj hur tecken ska skrivas ut. Vi kommer här bara kort beskriva Text, för en djupare presentation hänvisar vi till boken Interaktionsprogrammering i Smalltalk.

Form

Använder en sträng för teckenrepresentationen och något som kallas runArray för formatinformation.

Konstruktion

Text saknar litteral form men det är möjligt att konvertera en sträng till text och därmed utnyttja den litterala formen för String

```
enText := 'Detta är ett sätt att skapa en text' asText.
```

som ger en text utan speciell teckeninformation.

```
enFetText := 'En text i fetstil är det här' asText allBold.
```

Variabeln enFetText innehåller en text i fetstil.

```
kursivExempel := 'Ett kursivt ord' asText.
kursivExempel emphasizeFrom: 5 to: 11 with: #italic.
```

Här beskriver kursivExempel en text där ordet kursivt är kursivt.

Litteral form

Någon litteral form har inte Text, men man kan använda den litterala formen för String och därefter konvertera den till en Text med asText.

Konvertering

Meddelandet asText kan användas för att konvertera objekt som är subklasser till CharacterArray till text.

Övrigt

För att skriva ut texter så att teckensnitt mm syns så måste speciella klasser och metoder användas för presentationen, se boken Interaktionsprogrammering i Smalltalk. Menyalternativetet **print it** ger en utskrift utan formatteringar.

7.15 LinkedList och Link

En instans av LinkedList är en ordnad mängd i vilken varje ingående element pekar på sin efterföljare. Elementen i listan bör vara instanser av Link eller någon, troligen egendefinierad, subklass till denna.

Form

Använder en enkellänkad struktur där ett objekt ur LinkedList är listhuvud och element i listan är av typen Link eller subklasser till Link.

Konstruktion

Följande exempel konstruerar en LinkedList och lägger därefter in tre länkobjekt.

```
| collection |
collection := LinkedList new.
collection add: Link new; add: Link new; add: Link new.
collection
```

⇒*LinkedList (a Link a Link a Link)*

Instanser ur Link kan inte innehålla någon information. Om man vill att länkobjekten ska innehålla information så måste en subklass kontrueras där information kan läggas in.

Litteral form

Litteral form saknas och det är inte heller enkelt att utnyttja t ex Array's litterala form.

Konvertering

De vanliga metoderna för konvertering finns. Man måste dock tänka på att de objekt som läggs in i de nyskapade behållarna blir av typen Link eller en egendefinierad subklass.

7.16 Interval

Interval beskriver en ändlig aritmetisk utveckling.

En instans av Interval är en mängd av tal som beskriver en matematisk utveckling.

Form

Ett intervall är ett objekt med startvärde, slutvärde och steglängd.

Konstruktion

En instans av Interval kan tex skapas på följande sätt.

```
Interval from: 1 to: 3
```
⇒*(1 to: 3)*

Ett snabbt sätt att konstruera ett antal konsekutiva tal är att använda konstruktion start to: end. Om vi från detta tex vill konstruera en vektor i fallande ordning så behöver vi endast skicka reverse till intervallet och sedan ta hand om resultatet.

```
(1 to: 10) reverse
```
⇒*#(10 9 8 7 6 5 4 3 2 1)*

Det går också bra att konstruera ett intervall med angiven steglängd som i följande exempel.

```
(1 to: 10 by: 3) reverse
```
⇒*#(10 7 4 1)*

Indexgränserna eller steglängden får vara negativa, så följande uttryck fungerar utmärkt:

```
(-1.2 to: -2.3 by: -0.1) select: [:x | x fractionPart abs > 0.75]
```
#(-1.8 -1.9)

Litteral form

Någon litteral form finns inte men som vi sett kan man enkelt skapa intervall genom att skicka meddelanden till tal som då returnerar ett intervall.

7.17 ByteArray

Form

En ByteArray är en vektor som fungerar som en Array men med den skillnaden att elementen bara kan vara heltal från 0 till 255 dvs med 8-bitars komponenter.

Konstruktion

En instans av ByteArray skapas enklast med hjälp av dess litterala form #[]. Den fungerar på samma sätt som Array's litterala form. Det går också att skapa instanser med new:. För andra klasser initieras elementen till nil men eftersom ByteArray endast kan bestå av heltalen 0-255 så initieras alla element till 0 istället.

Litteral form

Nummertecken, vänster hakparentes, bytekod, höger hakparentes. Exempel:

```
#[12 23 0 123 245]
```

Övrigt

Lagrar data på ett kompakt och effektivt sätt. Därför kan denna klass tex användas då extrem snabbhet behövs eller då lite plats finns tillgänglig.

7.18 Exempel

Korta exempel

Nu ska vi ge några enkla exempel på hur behållarklasserna kan användas.

Exempel: Fakulteten på talet tal.

Med do:

```
| tal produkt |
tal := 5.
produkt := 1.
1 to: tal do: [:i| produkt := produkt * i].
```

```
produkt
```
⟹*120*

Med inject:into:

```
| tal |
tal := 5.
(1 to: tal) inject: 1 into: [:produkt : i | i * produkt]
```
⟹*120*

Först skapar uttrycket (1 to: tal) ett Interval. Därefter skickar vi inject:into: till intervallet. Reultatet av metodanropet blir den ackumulerade produkten.

Exempel: Teckenfrekvens i en sträng. Utskrift i frekvensordning.

```
| sorteradeOrd str |
str := 'Skriv kod som beräknar teckenfrekvens, och skriver ut tecknen sorterade i
frekvensordning'.
sorteradeOrd := SortedCollection sortBlock: [:k1 :k2 | (k1 at: 1) > (k2 at: 1)].
str asBag valuesAndCountsDo: [:tecken :förekomst |
                        sorteradeOrd add: (Array with: förekomst with: tecken)].
sorteradeOrd collect: [:array | array at: 2]
```
⟹*OrderedCollection ($e Character space $r $n $k $s $o $v $t $i $d $c $f $a $, $S $u $m Character cr $ä $h $g $b)*

Vi skapar en SorteredCollection med lämpligt sorteringsblock. Därefter omvandlar vi strängen str till en Bag. Sedan använder vi direkt Bag'en för att iterera över alla olika tecken i den. För att kunna sortera på antal förekomster av ett tecken så använder vi ett hjälpobjekt nämligen en Array som håller i antal förekomster av ett tecken och tecknet själv. Med hjälp av detta objekt kan vi ordna tecknen efter antal förekomster. Det sista som sker är att vi ur den sorterade listan skapar en ny ordnad lista av alla som bara består av tecknen, inte antal förekomster.

Exempel: Synonymlexikon där uppslagsord kan ha flera synonymer

```
| synonym |
synonym := Dictionary new.
synonym at: 'huvud'
        put: (Set with: 'skalle' with: 'boll' with: 'knopp' with:  'pall').
synonym at: 'näsa'
        put: (Set with: 'snok' with: 'nos' with: 'tryne' with: 'misslyckande').
synonym at: 'öga'
        put: (Set with: 'centrum').
synonym at: 'ansikte'
        put: (Set with: 'fejs' with: 'plyte' with: 'nia' with: 'plåt').
synonym
```
⟹*Dictionary ('huvud'->Set ('skalle' 'boll' 'knopp' 'pall') 'näsa'->Set ('snok' 'misslyckande' 'tryne' 'nos') 'ansikte'->Set ('plyte' 'plåt' 'fejs' 'nia') 'öga'->Set ('centrum'))*

Exempel: Lägg till synonym

```
synonym at: 'öga' put: ((synonym at: 'öga' ifAbsent: [Set new])
    add: 'korpglugg'; yourself).
```

Först plockas eventuella tidigare synonymer ut, om det inte finns några skapas en tom mängd. Därefter läggs den nya synonymen till mängden. Slutligen läggs mängden in i ordlistan.

```
(synonym at: 'ansikte' ifAbsent: [Set new]) includes: 'nos'.
```

⇒*false*

```
(synonym at: 'hår' ifAbsent: [Set new]) includes: 'barr'.
```

⇒*true*

```
synonym keys.
```

⇒*Set ('huvud' 'näsa' 'öga' 'ansikte')*

```
ord := synonym keys.
synonym do: [:set| ord addAll: set].
ord
```

⇒*Set ('boll' 'öga' 'pall' 'nos' 'ansikte' 'snok' 'tryne' 'skalle' 'huvud' 'korpglugg' 'knopp' 'nia' 'misslyckande' 'plyte' 'fejs' 'plåt' 'näsa' 'centrum')*

Kontrollsiffra

Nu ska vi konstruera en metod för att kontrollera att ett personnummers kontrollsiffra är riktig. Metoden returnerar ett booleskt värde som anger om siffran var riktig. Om den sista siffran i personnumret är utelämnad beräknas och returneras istället kontrollsiffran.

En kontrollsiffra i ett personnummer beräknas genom att multiplicera varannan siffra med ett, varannan med två och därefter räkna ut siffersumman av alla produkterna. Totalsumman ska adderat med kontrollsiffran bli jämnt delbart med tio.

Vi kan skriva följande metod i SequenceableCollection.

```
checkSum
    "Returnerar true om kontrollsiffran är riktig. Saknas kontrollsiffra räknas den
    ut och returneras"
    | sum |
    (self size between: 9 and: 10) "Rudimentär kontroll av gilltighet"
        ifFalse: [^false].
    sum := 0.
    1 to: 9 do: [:i | | factor product | "gå igenom alla siffror utom kontrollsiffran"
        factor := i odd
            ifTrue: [2]
            ifFalse: [1].
        product := (self at: i) digitValue * factor.
        product >= 10 ifTrue: [product := product \\ 10 + (product // 10)].
        sum := sum + product].
```



```
    ^self size = 9
        ifTrue: [10 - (sum \\ 10) \\ 10]
        ifFalse: [10 - (sum \\ 10) \\ 10 = self last digitValue]
```

Alternativt kan slingan som är markerad med "gå igenom alla siffror utom kontrollsiffran" skrivas om med hjälp av två slingor enligt:

```
2 to: 8 by: 2 do: [:i | sum := sum + (self at: i) digitValue].
1 to: 9 by: 2 do: [:j | | product |
        product := (self at: j) digitValue * 2.
        product >= 10 ifTrue: [product := product \\ 10 + (product // 10)].
        sum := sum + product].
```

där vi har unyttjat att endast siffror med udda index behöver multipliceras med en faktor 2. De tre sista raderna i metoden kan också skrivas (checkValue måste också läggas till de temporärdeklarerade variablerna):

```
checkValue :=
    sum = 0
        ifTrue: [0]
        ifFalse: [10 - (sum \\ 10)].
^self size = 9
    ifTrue: [checkValue]
    ifFalse: [checkValue = self last digitValue]
```

Vilken av dessa metoder som väljs beror kanske på smak men också på vad som uppfattas som mest lättläst.

Kryptering av strängar

I avsnitt 5.6 beskrev vi hur man med öppna nycklar kan kryptera och dekryptera heltalsvärden. Vi ska nu utöka PublicKeyRSA med några metoder som direkt från en sträng genererar en krypterad vektor samt omvänt givet ett krypterat meddelande dekrypterar och ger en sträng som resultat.

Vi börjar, för att strukturera problemet, med att konstruera en metod för att kryptera en vektor av heltalsvärden:

```
encryptArrayOfIntegers: integerArray
    ^integerArray collect: [:anInt | self encrypt: anInt]
```

Sedan konstruerar vi en metod som gör det omvända, dvs givet en krypterad vektor av heltal returnerar en ny vektor med dessa tal i dekrypterad form:

```
decryptArrayOfIntegers: integerArray
    ^integerArray collect: [:anInt | self decrypt: anInt]
```

I och med att vi har definierat metoderna i PublicKeyRSA så har instanser av PublicKeyRSA2 också direkt möjlighet att utnyttja dessa, då den

senare är subklass till den tidigare. Därför kan vi göra en liten test av det hela och använda den ”bättre” av de två klasserna, enligt tex följande:

```
rsa := PublicKeyRSA2 new.
"I detta fall blev r = 46372763159881, s = 6809779, vilka publiceras.
De hemliga värdena blev p = 6809743, q = 6809767och t = 34514437273687"
rsa encryptArrayOfIntegers: #(97 98 99 100).
⇒#(40733882154535 15934139810374 21763819796262 25685585840098 )
rsa decryptArrayOfIntegers:  #(40733882154535 15934139810374
21763819796262 25685585840098 )
⇒#(97 98 99 100 )
```

Nu skriver vi också de två metoderna som direkt ger en kryptering från en given sträng och vice versa.

```
encryptString: aString
    ^self encryptArrayOfIntegers: (aString asArray collect: [:ch | ch asInteger])
```

Där vi har utnyttjat collect: respektive asInteger för att samla ihop resultaten av att konvertera varje tecken i strängen till motsvarande numeriska presentation. Observera också att här behöver vi ”mellankonvertera” strängen till en vektor! Detta för att vi inte kan lagra heltalselement i en sträng!

Dekryptering blir lika enkelt genom att utnyttja att en sträng direkt kan konstrueras från heltalsvektor samt hjälpmetoden:

```
decryptAsString: integerArray
    ^String fromIntegerArray: (self decryptArrayOfIntegers: integerArray)
```

Nu kan vi slutligen testa det hela med vektorerna utbytta mot strängar.

```
rsa decryptAsString: (self encryptString: 'abcåääöd' )
⇒'abcåääöd'.
```

Sammanfattning

Termer

Behållare objekt som används för att lagra element.

Litteral form ett sätt att beskriva ett objekt direkt utan att skapa det via ett meddelande till en klass.

Smalltalk

Collection rotklass för alla behållare.

Behållare som definierar ordning mellan sina element är subklasser till SequenceableCollection

Enkelt att skapa behållare utgående från litteral form som #(a b c).

add- och remove-metoderna returnerar objektet, inte behållaren.

do: itererar över alla element.

Övningar

7.1 Vilka metoder definieras i OrderedCollection för att ta bort element?

7.2 Vi skulle kunna ha skrivit encryptString: på sidan 272 på följande sätt:

```
encryptString: aString
    ^self encryptArrayOfIntegers: aString asWordArray asArray
```

a) Varför fungerar det inte om vi utelämnar asArray?

b) Varför kan vi inte använda det unära meddelandet asByteArray om vi vill hantera strängar i vilka svenska tecken ingår?

7.3 I övning 4.9 konstruerade vi sköldpaddor av olika typer. Ett antal av dem kommer nu till en sluss. Skriv kod som hanterar att sköldpaddorna slussas igenom på ett rättvist sätt. Det skall vara möjligt att anropa metoder som skriver ut information i `Transcript`-fönstret om sköldpaddor i kön. Informationen kan bestå av exempelvis hur långt sköldpaddan har vandrat, om så är möjligt. Använd någon Collection för att implementera detta.

7.4 Skriv uttryck som på en kö med sköldpaddor beräknar total-, max- och medianvärden för tillryggalagd distans.

7.5 Gör en Collection som innehåller alla som gått längre än mediantillryggalagd distans.

7.6 Det finns många metoder för att numeriskt integrera en funktion. En av dem enklare och mer rättframma är Trapetsregeln. I Trapetsregeln utnyttjas sambandet

$$\int_a^b f(x)dx = h\left[\frac{1}{2}f(a) + f(a+h) + \ldots + f(a+(n-1)h) + \frac{1}{2}f(b)\right] + R \quad ,$$

där R är det fel som uppkommer vid de numeriska beräkningarna. Om vi, för enkelhets skull, negligerar R och skriver om högerledet som $h\left[\frac{1}{2}(f(a)+f(b)) + \sum_{}^{n-1} f(a+jh)\right]$ är det enkelt att använda Interval och block, för att beskriva regeln. Skriv ett block intAux := [:f :a :b :h | ...] som beräknar integralen av en funktion f given av ett block med en formell parameter, som tex funktionen $f(x) = \frac{1}{1+x}$ blir [:x | 1 / (1 + x)].

7.7 I ett berömt tvåpersonersspel genererar en spelare (A) ett mönster den andra spelaren (B) skall sedan försöka lista ut det aktuella mönstret. Därmed generar B en vektor med sin gissning. Alla komponenter i gissningen som överensstämmer med mönstret ger en svart pinne och alla element som finns med i mönstret, men inte exakt överenstämmer, ger en vit pinne. En del av spelet består i att A jämför en av Bs gissningar med mönstret.

En strategi för detta är:

1 för alla element som exakt överensstämmer
   - räkna upp den svarta räknaren
   - ta bort elementet från båda listorna
2 för alla element som ingår i mönstret gör följande
   - räkna upp den vita räknaren
   - ta bort elementet från gissningen

Implementera och testa detta i ett arbetsfönster.
Tips: använd en OrderedCollection. OrderedCollection>>removeAtIndex: tar bort element i viss position

8 Egna behållare

Detta kapitel besvarar bland annat följande frågor:

- Hur definierar man egna behållarklasser?
- Hur beskriver man länkade listor?
- Vad är ett listhuvud?
- Hur implementerar man en do:-metod?
- Vad är en variabel subklass?

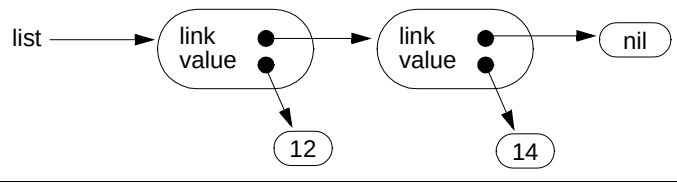
I föregående kapitel beskrevs behållarklasserna som följer med alla smalltalksystem. I det här kapitlet beskrivs hur man kan definiera egna behållarklasser som fungerar enligt samma principer som de i systemet. Detta ger även en fördjupad förståelse av systemets behållarklasser.

8.1 Länkade strukturer

Avsnittets främsta mål är att förklara principerna för hur behållare är konstruerade i Smalltalk. Vi gör detta genom att använda oss av olika typer av länkade listor. Vi illustrerar också sätt att utnyttja arvsmekanismer vid implementation av nya klasser.

Enkellänkad lista

Vi börjar med att konstruera en enkellänkad lista utan huvud.



Figur 8.1 Enkellänkad lista med elementen 12 och 14

```
Object subclass: #SimpleLinkedList
    instanceVariableNames: 'value link '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Variabla-Strukturer'
```

Åtkomstfunktioner för instansvariablerna value och link definieras på vanligt sätt, vi visar inte det här, men väl hur man använder dem för att skapa början till strukturen i figur 8.1.

```
| list link |
list := SimpleLinkedList new.
list value: 12.
link := list species new.
link value: 14.
list link: link
```

Vi definierar att ett element är sist i listan om länken är nil. Vi konstruerar en speciell metod, atEnd, för att testa detta.

```
atEnd
    ^self link isNil
```

Vi definierar att listan är tom om elementet är sist i listan och dess värde är nil.

```
isEmpty
    ^self atEnd and: [self value isNil]
```

Nu kan vi enkelt skriva size som returnerar antalet element i listan. Vi definierar size som en rekursiv metod. Det är nästan lika enkelt att skriva denna metod iterativt.

```
size
    ^self atEnd
        ifTrue: [self isEmpty
                ifTrue: [0]
                ifFalse[1]]
        ifFalse: [1 + self link size]
```

För att imitera beteende från OrderedCollection väljer vi att lägga till nya element i slutet av listan med addLast:.

```
addLast: aValue
    "Lägger till aValue sist i listan"
    | last |
    last := self.
    last isEmpty
        ifTrue: [last value: aValue]
        ifFalse: [[last atEnd]
                    whileFalse: [last := last link].
                last link: (self species new value: aValue)]
```

Metoden addLast: sätter den temporära variabeln last att peka ut aktuellt listelement. Därefter kontrolleras om listan är tom och i så fall läggs värdet in. Annars letar vi upp det sista elementet och konstruerar ett nytt element av samma typ som det aktuella objektet.

Av symmetriskäl skriver vi en metod för att ta bort element från slutet av listan.

```
removeLast
    "Tar bort sista elementet ur listan"
    | last lastToPreserve |
    last := self.
    last atEnd
        ifTrue: [last value: nil]
        ifFalse: [
            lastToPreserve := last.
            [last atEnd]
                whileFalse:
                    [lastToPreserve := last.
                    last := last link].
            lastToPreserve link: nil.
            lastToPreserve == self ifTrue: [lastToPreserve value: nil]]
```

Som vi såg i föregående kapitel så används metoden do: för att iterera över behållare av olika slag. Nu ska vi definiera do: så att den fungerar även på vår lista på vanligt sätt, dvs med ett block som argument. Blocket ska i sin tur vara definierat att ta ett element som argument, dvs de respektive elementen i listan.

```
do: aBlock
    "Utför aBlock på alla element"
    self isEmpty
        ifFalse: [aBlock value: self].          Applicera blocket på mottagaren
    self atEnd ifFalse: [self link do: aBlock]      fortsätt med nästa element
```

Vi testar do: genom att skriva ut värdena i listan:

```
| list tabs |
list := SimpleLinkedList new.
1 to: 3 do: [:i | list addLast: i].
tabs := 1.
list do: [:aLink | Transcript show: aLink value printString;
        crtab: tabs.
        tabs := tabs + 1].
Transcript cr
```

Vi använder crtab: för att få tabulerad utskrift. Utskriften blir:

```
1
    2
        3
```

Nu kan vi enkelt konstruera en stack genom att subklassa SimpleLinkedList. En stack måste naturligtvis ha metoderna push: och pop definierade. Dessa beskrivs enkelt med addLast: och removeLast.

```
SimpleLinkedList subclass: #SimpleStack
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Variabla-Strukturer'

push: x
    self addLast: x

pop
    self isEmpty
        ifTrue: [self error: 'Det går inte att ta bort element från en tom stack!'].
    ^self removeLast value
```

För att få pop att fungera som avsett måste vi implementera om removeLast så att det sista elementet returneras. I removeLast sätter vi value till nil för att markera att listan är tom och det kommer nu att ställa till problem. Ändrar vi removeLast till att returnera det sista elementet så kommer detta att vara samma element som refereras inifrån metoden och därmed kommer det returnerade värdet också bli nil när vi ska ta bort det sista elementet. Vilket är jättefel!

Vad har vi då för möjligheter att på ett enkelt sätt ordna upp detta, åtminstone i en testsituation? En lösning är att kopiera det borttagna elementet och returnera denna kopia istället för originalet. Ett annat sätt är att hålla en referens till det senast erhållna värdet och returnera detta istället för hela elementet från metoden.

Då detta inte är avsett att vara någon slutgiltig lösning och för att göra så få ingrepp som möjligt väljer vi att använda den föreslagna lösningen att omdefiniera removeLast i SimpleStack och returnera en kopia av det borttagna elementet.

För att kopiera ett objekt kan vi använda copy som alla objekt förstår.

```
removeLast
    "Tar bort sista elementet ur self och returnerar en kopia av det"
    | last lastToPreserve copiedElement |
    last := self.
    lastToPreserve := last.
    [last atEnd]
        whileFalse:
            [lastToPreserve := last.
            last := last link].
    copiedElement := last copy.
    lastToPreserve link: nil.
    lastToPreserve == self ifTrue: [lastToPreserve value: nil].
    ^copiedElement
```

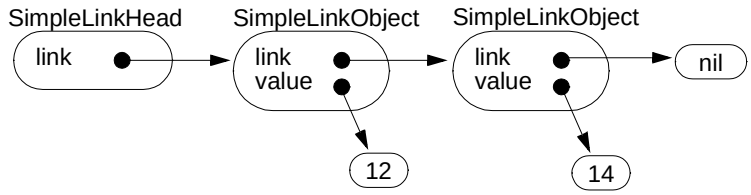

Nu testar vi stacken:

Kod	stack
| stack |	nil
stack := SimpleStack new.	()
stack push: 1.	(1)
stack pop.	()
stack push: 2.	(2)
stack push: 3.	(3 2)
stack pop.	(2)
stack pop	()

Figur 8.2 Stackoperationer

Enkelt länkad lista med huvud

Ett annat sätt att implementera en länkad lista är att skilja huvudet från värdeobjekten.



Figur 8.3 Enkellänkad lista med listhuvud

Genom att använda ett listhuvud så behöver man inte specialbehandla en tom lista, då den enbart består av sitt listhuvud. På det sättet blir det inga problem att ta bort det sista elementet ur, eller lägga till det första i, listan och vi behöver inte använda value = nil för att representera den tomma listan.

```
Object subclass: #SimpleLinkHead
    instanceVariableNames: 'link '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Variabla-Strukturer'
SimpleLinkHead subclass: #SimpleLinkedObject
    instanceVariableNames: 'value '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Variabla-Strukturer'
```

Vi beskriver inte de vanliga inspektorerna och mutatorerna för link och value.

```
linkObjectClass
    ^SimpleLinkedObject                    returnerar skönsvärdet på klassen
```

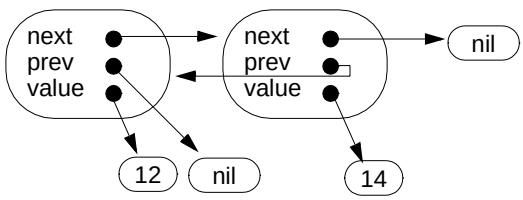
Nu väljer vi istället att implementera metoden addLast: som en rekursiv metod. Syftet med detta är endast att visa olika former av implementation av semantiskt likvärdiga metoder.

```
addLast: aValue
    "lägger till aValue sist i listan"
    self atEnd
        ifTrue: [self link: (self linkObjectClass value: aValue)]
        ifFalse: [self link addLast: aValue]

removeLast
    self atEnd ifFalse: [self link removeLast]

SimpleLinkHead>>isEmpty
    ^self link isNil

SimpleLinkedObject>>isEmpty
    "Eftersom det finns ett element är listan inte tom"
    ^false
```

Dubbellänkad lista

Enkellänkade listor har nackdelen att man endast kan gå åt ett håll i listan. Med en extra uppsättning pekare möjliggör man traversering åt bägge hållen.

```
Object subclass: #DoubleLinkedChain
    instanceVariableNames: 'value prev next '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Variabla-Strukturer'
```



Figur 8.4 Dubbellänkad lista med elementen 12 och 14

Nu definierar vi addAfter: som att objektet läggs in efter det objekt som är mottagare av meddelandet.

```
addAfter: aValue
    | newLink |
    newLink := self species value: aValue.
    self addLink: newLink

addLink: aDoubleLink
    "lägger till aDoubleLink efter self"
    | oldNext |
    oldNext := self next.
    aDoubleLink next: oldNext.
    oldNext notNil ifTrue: [oldNext prev: aDoubleLink].
    aDoubleLink prev: self.
    self next: aDoubleLink
```

Test:

```
| link |
link := DoubleLinkedChain value: 1.
link addAfter: 2.
```

Det fungerar bra så vi lägger till:

```
atEnd
    ^self next isNil

atStart
    ^self prev isNil

size
    "returnerar antalet element i listan"
    | sz prevCursor nextCursor |
    sz := 1.
    prevCursor := self prev.
    [prevCursor notNil]
        whileTrue:
            [sz := sz + 1.
            prevCursor := prevCursor prev].
            nextCursor := self next.
    [nextCursor notNil]
        whileTrue:
            [sz := sz + 1.
            nextCursor := nextCursor next].
    ^sz

remove
    self atEnd ifFalse: [self next prev: self prev].
    self atStart ifFalse: [self prev next: self next]
```

Test:

```
| link |
link := DoubleLinkedChain value: 1.
link add: 2.
link add: 3.
link next remove.
```

Ett problem är att det inte går att ta bort det sista objektet i kedjan och därmed få en tom lista! Detta problem kan återigen lösas genom att tex använda ett speciellt listhuvud eller med att det enda objektet i kedjan ersätts med ett nytt objekt som specifikt 'vet' att listan är tom.

För att förbereda för det senare av dessa alternativ så skriver vi en metod isEmpty som alltid returnerar false.

```
isEmpty
    ^false
```

Tanken är att en lista med element ur DoubleLinkedChain aldrig betraktas som tom. Om vi försöker ta bort det sista elementet så byter vi ut listan mot ett speciellt ”tomt” objekt som istället alltid svarar med true på isEmpty. En beskrivning av hur detta kan gå till finns nedan i DoubleLinkedChainEmpty.

Ändring av ett objekts klasstillhörighet

Vi ska nu illustrera ett dynamiskt sätt att hantera ett objekt som instans av olika klasser. Med denna teknik kan vi relativt enkelt konstruera instanser som byter klasstillhörighet, i detta fall beroende av om listan är tom eller ej, inkluderande bevarande av eventuella referenser till det aktuella objektet. Kärnan i denna teknik är metoden become:, se sidan 222, vars väsentliga delar tillhandahålls direkt av Smalltalks virtuella maskin. Det är viktigt att komma ihåg att metoden become: liksom en del andra ”system-metoder” bör användas väldigt restriktivt och först efter det att andra alternativ har förkastats!

```
Object subclass: #DoubleLinkedChainEmpty
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Variabla-Strukturer'
```

Då en instans av DoubleLinkedChainEmpty enligt vår beskrivning alltid ska vara tom så blir metoden isEmpty väldigt enkel.

```
isEmpty
    ^true
```

Av symmetriska skäl väljer vi att också implementera en metod remove med tom kropp i DoubleLinkedChainEmpty. I och med att metoden är definierad på det sätt som följer så händer ingenting annat än att en referens till mottagaren av meddelandet returneras från metoden.

```
remove
    "Do nothing"
```

Vi konstruerar också en metod isOnlyObject i DoubleLinkedChain vars syfte är att returnera true om listan består av endast ett objekt.

```
isOnlyObject
    ^self atStart and: [self atEnd]
```

En alternativ implementation av metoden vore att använda size och undersöka om listans storlek är lika med ett. Nackdelen med detta lösningsförslag är att den är mindre effektiv då vi alltid måste traversera hela listan (vilket size gör) innan vi vet om den består av ett element eller inte.

För att få önskat byte av klasstillhörighet då det sista objekt i listan tas bort så måste vi också definiera om metoden remove i DoubleLinkedChain.

```
DoubleLinkedChain>>remove
    "Tar bort ett element ur listan"
    self isOnlyObject
        ifTrue: [self become: DoubleLinkedChainEmpty new]
        ifFalse:  [  self atEnd
                        ifFalse: [self next prev: self prev].
                    self atStart ifFalse: [self prev next: self next]]
```

Vi beskriver också hur listan ska presenteras textuellt på skärmen.

```
printOn: aStream
    "Skriver ut hela listan"
    aStream nextPut: ${.
    self printValuesOn: aStream.
    aStream nextPut: $}

printValuesOn: aStream
    "Skriver ut elementen i listan separerade med blanktecken"
    self value printOn: aStream.
    self atEnd
        ifFalse:  [aStream nextPut: Character space.
                   self next printValuesOn: aStream]
```

Vi måste också se till att vi byter från tom lista till ett ”vanligt” listelement om vi försöker lägga till något till den tomma listan.

```
DoubleLinkedChainEmpty>>add: aValue
    self become: (DoubleLinkedChain value: aValue)
```

För att få ett presentationsätt motsvarande icke tomma listor får vi inte heller glömma att definiera om printOn: i DoubleLinkedChainEmpty.

```
printOn: aStream
    aStream nextPutAll: '{}'
```

Med dessa definitioner kan vi pröva om det hela fungerar genom att tex exekvera följande kodavsnitt:

```
| link |
link := DoubleLinkedChain value: 1.
link add: 2.
link next remove.                    "link är en DoubleLinkedChain"
link remove.                         "link blir en DoubleLinkedChainEmpty"
link isEmpty.                        "ger true"
link add: 3.
link add: 4.
```

⇒*{3 4}*

Därefter skriver vi removeLast,

```
removeLast
    "tar bort sista elementet i listan"
    | candidate |
    candidate := self.
    [candidate atEnd]
        whileFalse: [candidate := candidate next].
    candidate remove

DoubleLinkedChainEmpty>>removeLast
    self remove
```

Nu prövar vi det hela genom att utföra följande kodavsnitt:

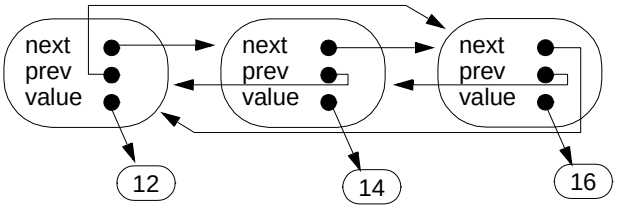
```
| link |
link := DoubleLinkedChain value: 'start'.
1 to: 3 do:  [:i| Transcript show: link printString; cr.
              link add: i].
[link isEmpty]
    whileFalse: [link removeLast.
                 Transcript show: link printString; cr].
1 to: 3 do:  [:i| Transcript show: link printString; cr.
link add: i].
```

vilket ger följande utmatning i utskriftsfönstret:

```
{'start'}
{'start' 1}
{'start' 2 1}
{'start' 3 2}
{'start' 3}
{'start'}
{}
{}
{1}
{1 2}
```

Cirkulär lista

Som ett sista exempel på listor ger vi en beskrivning av hur en dubbellänkad cirkulär lista utan speciellt listhuvud kan konstrueras.



Figur 8.5 Dubbelt länkad cirkulär lista

I denna beskrivning implementerar vi också motsvarigheten till några av de viktigaste styrstrukturmetoderna som Smalltalk använder i sina fördefinierade klasser av Collection-typ.

```
Object subclass: #DoubleLinkedCircularChain
    instanceVariableNames: 'value prev next '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Variabla-Strukturer'

KLASSMETODER

value: aValue prev: prevLink next: nextLink
    "Skapar en ny instans och initierar länkarna"
    ^self basicNew
        value: aValue
        prev: prevLink
        next: nextLink

new
    ^self basicNew initialize

value: aValue
    ^self new value: aValue

INSTANSMETODER

initialize
    prev := next := self
```

De olika åtkomstmetoderna skrivs på vanligt sätt och en 'högre ordningens' metod för att samtidigt sätta värde och de båda länkarna kan skrivas på följande sätt:

```
value: aValue prev: prevLink next: nextLink
    self value: aValue.
    self prev: prevLink.
    self next: nextLink.
```

Denna metod ger möjligheten att enkelt skapa nya listelement samt ange deras värden och pekare.

```
add: aValue
    "Lägger till aValue efter self"
    | newElement |
    newElement := self species
        value: aValue
        prev: self
        next: self next.
    self next prev: newElement.
    self next: newElement.
    ^newElement
```

Vi definierar do: som applicerar ett block på alla element i listan.

```
do: aBlock
    "Utför aBlock på alla element i listan"
    | current |
    current := self.
    aBlock value: self.
    [(current := current next) ~~ self]
        whileTrue: [aBlock value: current]
```

Metoden size nedan returnerar längden på listan. Den utnytjar metoden do:, där den för varje element enbart räknar upp variabeln tally.

```
size
    | tally |
    tally := 0.
    self do: [:each | tally := tally + 1].
    ^tally
```

För att implementera utskriftsmetoden printOn: används också metoden do: så att alla element får skriva ut sitt värde. Om vi skriver ut en ordnad lista som bestående av talen ett till tio så blir resultatet

⟹*{1 2 3 4 5 6 7 8 9 }*

```
printOn: aStream
    "Skriver ut hela listan"
    aStream nextPut: ${.
    self do: [:each |
                each value printOn: aStream.
                aStream nextPut: Character space].
    aStream nextPut: $}
```

Metoden collect: returnerar som vanligt en ny lista med resultatet då ett block har utförts på alla element.

```
collect: aBlock
    "returnerar en lista med aBlock utfört på alla element"
    | element start |
    self do: [ :each |
            element isNil
                ifTrue: [start := element := self species value: (aBlock value: each)]
                ifFalse: [element := element add: (aBlock value: each)]].
    ^start
```

Att vi använder element för att peka ut aktuellt element och start för att referera till första elementet i listan beror delvis på att vi i DoubleLinkedCircularChain endast har definierat metoder för att lägga till nya element *efter* utpekat element (och alltså inte sist vilket vi skulle vara betjänta av här).

Vidare undersöker vi varje gång i do-loopen om någon resultatlista är skapad. Om så är fallet lägger vi till nya element till listan och annars skapar vi en ny lista med detta element som enda medlem. Det senare behövs då vi inte har implementerat något sätt att hantera tomma listor. Vi har också utnyttjat att variabler från början är nil. Med ett listhuvud och addLast: skulle collect: vara betydligt enklare.

Nu kan vi pröva collect: genom att kvadrera alla element i en lista.

```
(DoubleLinkedCircularChain value: 5)
        collect: [:x | x value squared]
```

⟹ *{25 }*

Vi behöver också vara försiktiga då vi konstruerar nya listor. Dvs vi måste vara medvetna om att vi inte kan skapa tomma listor samt att element endast läggs till efter visst utpekat element i listan.

Om vi vill skapa listan {1 2 3 4 5 } och konstruera en ny lista med kvadraterna av listans komponenter kan vi gå till väga på följande sätt:

```
| list element |
list := (DoubleLinkedCircularChain value: 1).
element := list.
(2 to: 5) do: [:i | element := element add: i]. "list är nu {1 2 3 4 5 }"
list collect: [:x | x value squared]
```

⟹ *{1 4 9 16 25 }*

Nu konstruerar vi detect: aBlock ifNone: failureBlock som ska returnera första element som aBlock blir sant för. Saknas element i listan som blocket blir sant för så returneras resultatet av att failureBlock utförs.

```
detect: aBlock ifNone: failureBlock
    self do: [:each | (aBlock value: each)
        ifTrue: [^each]].
    ^failureBlock value
```

Metoden detect: genererar ett felavbrott om den inte hittar något element.

```
detect: aBlock
    ^self detect: aBlock ifNone: [self error: 'Inget sådant element!']
```

Test:

```
| list element result |
list := (DoubleLinkedCircularChain value: 0).
element := list.
(2 to: 5) do: [:i | element := element add: i].
result := list detect: [:x | x value odd].
result value
```

⇒*3*

Då styrstrukturer blir tillkrånglade med denna cirkulära lista spar vi resten av styrstrukturerna som övningar för den intresserade läsaren.

Som framgår är ingen av de beskrivna klasserna eller dess implementationer fullständiga. Ändå beskriver de vissa centrala mekanismer och visar på vissa möjliga tekniker som ska ingå vid en fullständig implementation. Utan större ansträngning bör de kunna fyllas på med flera operationer, men detta ger inget väsentligt nytt för framställningen. Vidare har vi ännu inte illustrerat användning av tex abstrakta klasser för att på ett mer generiskt sätt konstruera en bas för olika former av mängder genom ytterligare generalisering eller specialisering.

Några av de ovan beskrivna problemen leder in oss på nästa avsnitt där vi visar hur variabla klasser kan konstrueras.

8.2 Variabla klasser

Att använda länkade listor är inte alltid effektivt. Nu ska vi beskriva hur instanser med variabel storlek kan kontrueras. Objekt med variabel storlek, tex Array och OrderedCollection, har fördelar som:

- Snabb åtkomst av element vid givet index tex då at: används.
- Spar minne eftersom inga extra pekare behövs.

Det finns också nackdelar med dessa klasser jämfört med listor:

- Långsamt eller omöjligt, beroende på klass, att lägga till objekt mitt i ordnade behållare.
- Långsammare än listor om storleken ofta förändras.

Klassdefinition

De finns i huvudsak två olika klasser av variabel typ. I den första och vanligaste kommer komponenterna bestå av indicerbara variabler av pekartyp dvs objekt ur vanliga klasser som man själv kan definiera.

En klass av denna typ konstrueras genom användning av följande metod:

```
Object variableSubclass: #VariableClass
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Variabla-Strukturer'
```

Den andra möjliga variabla klasstypen har också indexerbara variabler men av icke pekartyp där varje element är en byte.

```
Object variableByteSubclass: #VariableByteClass
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Variabla-Strukturer'
```

Av dessa två är det den första som används vid konstruktion av så gott som alla grundläggande klasser avsedda att hantera behållare. Den byte-orienterade formen används bland annat för kommunikation med externa enheter (tex filer) och för kommunikation mellan datorer (tex via en port).

Instansiering

En instans av en variabel klass kan skapas med new. Men på detta kommer endast en instans utan komponenter att konstrueras. För att skapa ett objekt med given storlek finns det en metod new: som konstruerar ett objekt med angivet antal komponenter. Exempel:

```
variableObject := VariableClass new: 5
```

Liksom new har en ekvivalent metod basicNew för att konstruera nya instanser så har new: metoden basicNew: för konstruera indicerbara instanser, så vi kan också skapa variableObject på följande sätt:

```
variableObject := VariableClass basicNew: 5
```

Metoderna basicNew respektive basicNew: ska *aldrig* omdefinieras i någon subklass! Dvs alla objekt ska alltid ha direkt tillgång till de ursprungliga metoderna basicNew och basicNew: som finns definierade i Behavior!

Alla element i en instans av en variabel klass av pekartyp kommer att bli nil. Så variableObject kommer se ut som följer:

```
#(nil nil nil nil nil )
```

I en instans av en byteorienterad klass kommer elementen däremot automatiskt att sättas till noll, dvs

```
variableObject := VariableByteClass new: 5.
⇒#[0 0 0 0 0 ]
```

Åtkomst

Alla variabla klasser bygger på att systemet tillhandahåller metoder för att konstruera indicerade klasser och i huvudsak följande två metoder för att referera respektive tilldela värde till ett element vid ett givet index:

at:

at:put:.

Dessa två metoder definierade i Object och meddelanden med dessa namn kan därför skickas till alla typer av objekt. De har normalt ingen mening för klasser av icke indicerbar typ och kommer då ofta att resultera i ett felavbrott.

Metoderna at: och at:put: är så viktiga att alla subklasser alltid ska kunna komma åt deras respektive orginalmetoder, definierade i Object, att Object också tillhandahåller följande metoder:

basicAt:

basicAt:put:

Dessa två metoder är helt ekvivalenta med Object>>at: respektive Object>>at:put: och ska *aldrig* omdefinieras i någon subklass! Detta är analogt med basicNew och basicNew: som alternativ till new och new:.

8.3 Behållare med fast storlek

Som första större exempel på en variabel klass ska vi konstruera en klass av vektortyp som hanterar variabla objekt.

Vektor

En vektor är ett objekt som ges en fast storlek i vilken komponenter kan läggas in och läsas med heltalsindex i intervallet 1 (ett) t o m storleken av vektorn.

För att illustrera de mest grundläggande mekanismerna vid konstruktion av en klass av variabel typ konstruerar vi en klass Vector.

```
Object variableSubclass: #Vector
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Variabla-Strukturer'
```

Efter denna definition kan vi omedelbart konstruera nya instanser av klassen med new: där argumentet anger önskad storlek:

```
| vector |
vector := Vector new: 10.
vector size
```

⇒*10*

Vi kan också läsa respektive modifiera innehållet, genom att använda at: respektive at:put:. Vi prövar den nya klassen genom att utföra följande kodavsnitt:

```
| vector |
vector := Vector new: 3.
vector at: 1 put: 'ett'; at: 2 put: 2; at: 3 put: 46.
vector inspect
```

Nu definierar vi den mest centrala styrstrukturen för variabla objekt, nämligen:

```
do: aBlock
    1 to: self size do: [:i | aBlock value: (self at: i)]
```

Vad vi har gjort är att utnyttja möjligheten att fråga ett objekt om dess storlek (size) och Smalltalks sätt att skriva en slinga från givet startvärde tom slutvärde och för varje varv i slingan utföra argumentblocket (start to: stopp do: enArgumentsBlock). Lägg märke till hur elementen hämtas med self at: i.

Nedan konstruerar vi en vektor med tre komponenter, ger de indicerade variablerna värdet 10, 1 respektive 100. Vi använder oss av vectorSum som initieras till noll för att slutligen utnyttja do: till att summera alla vektorns värden.

```
| vector vectorSum |
vector := Vector new: 3.
vector at: 1 put: 10; at: 2 put: 1; at: 3 put: 100.
vectorSum := 0.
vector do: [:each | vectorSum := vectorSum + each].
vectorSum
```

⇒*111*

Vi konstruerar också metoden collect: som i princip blir ekvivalent med do:. Den väsentliga skillnaden är att vi använder en resultatvektor där vi placerar resultatet av att utföra aBlock på varje element i vektorn.

```
collect: aBlock
    "returnerar en lista med aBlock utfört på alla element"
    | newVector |
    newVector := self species new: self size.
    1 to: self size do: [:i | newVector at: i put: (aBlock value: (self at: i))].
    ^newVector
```

För att ge instansen en anpassad utskrift definierar vi om metoden printOn: (som används vid **print it** av objekt). Med följande definition av printOn: kommer en instans av vektor presentera sig på formen vänster

krullparantes ({), elementen i vektorn åtskilda med en blank och slutligen höger krullparantes (}). Observera hur vi använder den ovan skrivna do:–metoden samt skickar printOn: till respektive element vilket medför att de själva får avgöra hur de ska presentera sig.

```
printOn: aStream
    aStream nextPut: ${.
    self do:  [:element |
        element printOn: aStream.
        aStream space].
    aStream nextPut: $}
```

Nu testar vi collect: och printOn: med följande kodexempel:

```
| vector |
vector := Vector new: 3.
vector at: 1 put: 10; at: 2 put: 1; at: 3 put: 100.
vector collect: [:each | each squared]
```

⟹*{100 1 10000 }*

Nu konstruerar vi metoder för att testa på förekomster av element i Vector.

Vi börjar med att skriva metoden includes:.

```
includes: anObject
    self do: [:each | anObject = each ifTrue: [^true]].
    ^false
```

Vi använder do: för att iterera över alla element i vektorn, jämför det aktuella elementet (each) med argumentet anObject och om likhet är uppfylld returneras true från metoden. Om vi däremot efter att ha gått igenom alla element inte har hittat något som är lika med anObject så returneras false.

Ofta vill man veta om en behållare är tom. Metoder som svarar på det brukar heta isEmpty och kan implementeras på följande sätt:

```
isEmpty
    ^self size = 0
```

Vi skriver också en metod som ger antal förekomster av ett visst objekt.

```
occurrencesOf: anObject
    "Returnarar antalet förekomster av anObject"
    | tally |
    tally := 0.
    self do: [:each | anObject = each ifTrue: [tally := tally + 1]].
    ^tally
```

Observera att vi har använt värdelikhet mha '=' och inte referenslikhet med '==' vilket medför att två objekt kan vara lika utan att de pekar på samma objekt, se avsnitt 3.1.

Detta leder oss in på att definiera vad vi menar med likhet av två vektorer. Gör vi inget åt '=' så kommer Object>>= att användas och det vill vi inte då den är ekvivalent med referenslikhet, '==' . Vi väljer att ge en lite vidare form av '=' i Vector, nämligen att de vektorer vi vill jämföra är av samma typ (species), har samma storlek samt att alla element är '='–lika. Då kan vi definiera metoden = på följande sätt:

```
= otherCollection
    | size |
    self species == otherCollection species ifFalse: [^false].
    (size := self size) = otherCollection size ifFalse: [^false].
    1 to: size do: [:index | (self at: index) = (otherCollection at: index)
                                                    ifFalse: [^false]].
    ^true
```

Metoden detect:ifNone: kan också enkelt implementeras mha do:

```
detect: aBlock ifNone: exceptionBlock
    self do: [:each | (aBlock value: each)
            ifTrue: [^each]].
    ^exceptionBlock value
```

För att frigöra klienter från onödigt bekymmer med felhantering skriver vi också:

```
detect: aBlock
    ^self detect: aBlock ifNone: [self notFoundError]
```

Där vi definierar notFoundError som följer:

```
notFoundError
    self error: 'Elementet fanns inte i mängden!'
```

I kapitel 12 ska vi visa hur man på ett mer strukturerat sätt kan fånga och hantera fel. Vi prövar de nya metoderna:

```
| vector |
vector := Vector new: 4.
vector at: 1 put: 10; at: 2 put: 8; at: 3 put: 6; at: 4 put: 5. " {10 8 6 5 }"
vector detect: [:each | each < 8]
```

⇒*6*

För att ge användare möjligheter till att kopiera hela eller delar av en vektor tillhandahåller vi också:

```
copyFrom: start to: stop
    | newSize newVector |
    newSize := stop - start + 1.
    newVector := self species new: newSize.
    1 to: newSize do: [:i | newVector at: i put: (self at: start - 1 + i)].
    ^newVector
```

Om vi tex vill kopiera andra tom tredje komponenten av en vektor så kan vi nu enkelt göra detta:

```
| vector |
vector := Vector new: 4.
vector at: 1 put: 10; at: 2 put: 8; at: 3 put: 6; at: 4 put: 5.
vector copyFrom: 2 to: 3
```

⇒ *{8 6 }*

Metoden select: vars avsikt är att välja ut alla element som uppfyller villkoren i blocket som ges som argument, kan skrivas:

```
select: aBlock
    "returnerar en vektor med alla element som uppfyller aBlock"
    | newVector insertIndex |
    newVector := self species new: self size.
    insertIndex := 0.
    1 to: self size do: [:index |
        (aBlock value: (self at: index))
            ifTrue: [newVector at: (insertIndex := insertIndex + 1)
                                put: (self at: index)]].
    ^newVector copyFrom: 1 to: insertIndex
```

Nu testar vi återigen det hela genom att konstruera en likadan vektor som i de två tidigare exemplen, fast nu använder vi select: istället. Här väljer vi ut alla komponenter som är udda.

```
| vector  |
vector := Vector new: 4.
vector at: 1 put: 10; at: 2 put: 8; at: 3 put: 6; at: 4 put: 5.
vector select: [:each | each odd]
```

⇒*{5}*

Vi avslutar med att skriva metoden för reject:, som gör det omvända mot select:, dvs samlar ihop alla element som *inte* uppfyller villkoret.

```
reject: aBlock
    ^self select: [:element | (aBlock value: element) == false]
```

Vi prövar reject: med hjälp av följande kod:

```
| vector |
vector := Vector new: 4.
vector at: 1 put: 10; at: 2 put: 8; at: 3 put: 6; at: 4 put: 5.
vector reject: [:each | each < 100]
```

Eftersom alla element uppfyller villkoret blir resultatet en tom vektor.

8.4 Dynamiska mängder

I många tillämpningar kan det vara svårt att redan från början veta hur stort utrymme som kommer att fordras vid exekveringen av ett program. I dessa fall kan man ofta ta till ett tillräckligt stort utrymme för att klara även eventuella extremfall. Denna lösning är inte tillfredsställande av främst två skäl:

1) De fall då endast en del av det allokerade utrymmet utnyttjas slösar med minne.

2) Det kan behövas mer utrymme än vad som antogs vara maximalt!

Det senare går i vissa fall att lösa genom att tex fråga användaren om utrymmesbehov. Det finns en bra lösning på båda dessa problem, nämligen att använda länkade listor. Nackdelen med länkade listor är att man kan få långa åtkomsttider som följd och delvis slösar med utrymme.

Använder man sig av trädstrukturer så blir åtkomsten snabb, men man slösar fortfarande med utrymme. Hur stort slöseriet är beror på hur stora elementen är. Om elementen är på 200 kB, så är det fånigt att påstå att två extra pekare per element skulle vara en belastning.

Det finns andra lösningar som både sparar på utrymme och ger snabb åtkomst. I nästa avsnitt ska vi främst ägna oss åt tekniker som kombinerar dynamisk tillväxt med snabb åtkomst.

DynamicVector

I många situationer är det intresserant av att både kunna deklarera en mängd statiskt, som i fallet Vector ovan, men ändå erbjuda en möjlighet att lägga till element på ett dynamiskt sätt.

```
Vector variableSubclass: #DynamicVector
    instanceVariableNames: 'lastIndex '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Variabla-Strukturer'
```

Vi definierar helt enkelt en DynamicVectors storlek som värdet av lastIndex, och alltså inte som hela dess kapacitet. Därför skriver vi över superklassens size med följande metod:

```
size
    ^lastIndex
```

Det kan fortfarande vara intressant för klienter att fråga om vektorns maximala kapacitet. Metoden capacity returnerar vektorns storlek.

```
capacity
    ^self basicSize
```

För att garantera initieringen av vektorns kapacitet till ett heltalsvärde så skriver vi också om klassmetoden new: enligt följande:

```
new: anInteger
    ^(super new: anInteger) setIndices
```

Nu måste vi också skriva instansmetoden setIndices (som används av new: ovan). Då vektorn alltid är tom från början så sätter vi helt enkelt instansvariabel lastIndex till 0.

```
setIndices
    lastIndex := 0
```

Vår förenklade vektor kan enbart växa i storlek, inte minska.

För att hela tiden uppdatera lastIndex så måste också metoden at:put: skrivas om. Lägg märke till hur vi använder oss av superklassens at:put:.

```
at: index put: value
    lastIndex := lastIndex max: index.
    ^super at: index put: value
```

Nu kan vi testa det hela genom att som nedan konstruera en vektor med kapaciteten åtta, men bara utnyttja de fyra första indexen.

```
| vector |
vector := DynamicVector new: 8.
vector at: 1 put: 10; at: 2 put: 8; at: 3 put: 6; at: 4 put: 5.
vector size
```

⇒*4*

Metoden collect: från sidan 293 fungerar också omedelbart.

```
| vector |
vector := DynamicVector new: 8.
vector at: 1 put: 10; at: 2 put: 8; at: 3 put: 6; at: 4 put: 5.
vector collect: [:each | each odd]
```

⇒ *{false false false true }*

Eftersom vi har en sista position i klassen som räknas upp då element läggs till så är det enkelt att definiera metoden för att lägga till element i slutet av DynamicVector:

```
addLast: anElement
    lastindex := lastindex + 1.
    self at: self size put: anElement.
```

Först räknar vi upp lastindex, därefter placeras anElement på sista plats.

Styrstrukturer

I och med att vi har definierat DynamicVector som subklass till Vector fungerar omedelbart alla styrstrukturer som är definierade i Vector. Att de fungerar inser vi genom att betrakta de metoder som är omdefinierade i DynamicVector.

Först och främst är at:put: omdefinierad så att den kontrollerar om det givna indexet är större än det tidigare givna största indexet. Om så är fallet så uppdateras lastIndex, vilken sedan används som värde på den i DynamicVector omdefinierade metoden size. Vidare använder metoden Vector>>do: meddelandet size för att ta reda på en vektors storlek. Slutligen är collect:, detect:ifNone:, detect:, select: samt reject: alla konstruerade med hjälp av do:. Det senare gör att även dessa metoder fungerar smärtfritt för instanser av DynamicVector.

8.5 Abstrakt mängdbeskrivning

De olika vektorklasserna har stora likheter. Vi kan utnyttja detta för att både ge bättre lokalitet av koden, förenkla specialiseringar, generaliseringar, förbättra läsbarheten och förenkla förståelsen av denna typ av klasser.

Klassen OrderedContainer

Som exempel på en abstrakt mängdklass definierar vi OrderedContainer. Klassen motsvarar Smalltalks klass SequenceableCollection, men vår är starkt förenklad. Den definierar behållare i vilka man kan lägga till element med add-metoder. Elementen ordnas i behållaren. Vi kan även definiera test- och iterations-metoder i den abstrakta klassen. OrderedContainer kommer att ha VariableVector och SimpleLinkedList som subklasser. I den abstrakta klassen OrderedContainer definierar vi de delar som de båda subklasserna har gemensamt. Med en abstrakt klass underlättar vi också för eventuella framtida behållarklasser som kan utnyttja mycket av koden från OrderedContainer.

Testmetoder

I en hierarki av klasser behöver man ofta avgöra om ett objekt tillhör en viss klass så att man vet vilka egenskaper det har. Det kan kontrolleras genom att testa på klasstillhörighet tex med isKindOf:. Men om man gör det så skapar man ett beroende av en klass, istället för ett beroende till en egenskap. Exempel på sådana metoder är frågan om ett objekt har en maxstorlek, hasMaxSize. För SimpleLinkedList finns det ingen maxstor-

lek medan VariableVector har en maxstorlek. Om vi antar att de flesta klasserna i hierarkin har maxstorlek så blir definitionen:

```
OrderedContainer>>hasMaxSize
    "svarar på om konkreta subklasser har maxstorlek"
    ^true
```

SimpleLinkedList saknar maxstorlek så där definieras hasMaxSize om:

```
SimpleLinkedList>>hasMaxSize
    "jag har ingen maxstorlek"
    ^false
```

Lägga till element

Det finns många olika metoder för att lägga till element i en behållare. Några metoder är addLast:, add: och addAll:. Metoden add: lägger till ett element på någon för klassen lämplig position. För OrderedContainer är det naturligt att låta metoden add: lägga till elementet sist. Metoden addAll: lägger argumentets alla element till mottagaren.

Eftersom DynamicVector och SimpleLinkedList till sin inre struktur är helt olika så måste dessa klasser själva implementera en del grundmetoder. Vi väljer att definiera metoden addLast: anElement i subklasserna. Nu kan vi i den abstrakta klassen beskriva både add: och addAll: i termer av addLast:

```
OrderedContainer>>add: anElement
    "lägg till elementet anElement på lämplig plats.Vi antar att sista position är en
    lämplig plats"
    self addLast: anElement
```

Vi låter metoden add: direkt anropa addLast:. Definitionen av metoden addAll: blir:

```
OrderedContainer>>addAll: aCollection
    "Lägger till alla element i aCollection i mig"
    aCollection do: [:anElement | self add: anElement]
```

För varje element i aCollection så lägger vi till elementet med metoden add:. Nu kvarstår problemet med hur vi ska implementera metoden addLast: i OrderedContainer. Den ska vara definierad i subklasserna, men det är bra att markera i den abstrakta klassen att man förväntar sig att en viss metod ska implementeras i subklasser.

```
OrderedContainer>>addLast: anElement
    "lägger anElement sist bland alla element i mottagaren"
    self subclassResponsibility
```

Som vanligt markerar subclassResponsibility att subklasser ska omdefiniera metoden. Om inte addLast: omdefinieras så kommer anrop av den och därmed även add: och addAll: att orsaka felavbrott.

Iterationsmetoder

För att beskriva iterationsmetoder som do:, collect: och select: så söker vi en kärna av metoder som vi kan utgå ifrån för att definiera resten av iterationsmetoderna. Här räcker det med att definiera do: i subklasserna och sedan kan vi utnyttja do: för de övriga itererande metoderna.

```
Vector>>collect: aBlock
    "Samlar ihop resultaten av aBlock, när det utförs med alla mina element."
    "typen på resultatet blir detsamma som min typ"
    | result index|
    result := Array new: self size.
    index := 1
    self do: [:element |
            result at: index put: (aBlock value: element).
            index := index +1].
    ^self species withAll: result.
```

Metoden förutsätter endast att klassen är variabel, att den förstår do: och att man kan skapa nya objekt med withAll:.

8.6 Konkreta subklasser

I de konkreta klasserna ska vi implementera de delar som inte kan delas med andra klasser. Antingen för att funktionaliteten är unik för klassen eller därför att metoden är beroende av datastrukturen som objektet använder. Ytterliggare en orsak att implementera metoder i konkreta subklasser är att man där kan utnyttja den inre strukturen på objektet till att skriva effektivare kod.

Om vi studerar metoden collect: från avsnitt 8.5 så ser vi att den temporärt använder en Array för att lagra resultaten av blockanropen. Därefter skapas en behållare i vilken resultaten läggs in. Om vi nu utnyttjar det vi känner till om VariableVector så kan vi skriva om metoden collect:

```
VariableVector>collect: aBlock
    "returnerar en lista med aBlock utfört på alla element"
    | answer |
    answer := self species new: self size.
    1 to: self size do: [:i| answer at: i put: (aBlock value: (self at: i))]
```

Här utnyttjar vi att VariableVector förstår at: och at:put: och därför behöver vi inte använda den temporära vektorn vi hade tidigare.

Effektivitet

I exemplet med SimpleLinkedList och VariableVector så sker det en del ”onödiga” metodanrop eftersom kod är skriven i den abstrakta klassen Orde-

redContainer. Ett exempel är metoden addAll: som anropar add: som i sin tur anropar addLast: för varje element som ska läggas till. Det blir därmed två extra metodanrop för varje element som ska läggas till. Fördelen är att genom att utgå från relativt få metoder som är beroende av datastrukturen så minskar man risken för fel och tiden för felsökning minimeras.

Det finns speciella verktyg för optimering av koden. Med dess hjälp kan man enkelt se var mest tid av exekveringen verkligen sker. Tips: Skriv först koden så att den fungerar. När den fungerar, undersök om det är nödvändigt att optimera och gör det i så fall på rätt ställen.

8.7 Egendefinierd subklass till Collection

När man behöver en ny typ av behållare av generellt slag kan det ofta vara bra att låta den vara subklass till någon av Collection-klasserna så att den får samma gränssnitt som de övriga behållarna i systemet. På det sättet får man också mycket kod som redan är avlusad i superklasserna. Endast några få metoder behöver skrivas för den nya klassens specialla beteende. Vi ger här ett exempel på utökning av Collection-hierarkin.

Subklass till Dictionary

Antag att vi vill definiera en subklass till Dictionary som fungerar på följande sätt:

- Det externa *gränssnittet* ska vara detsamma som för Dictionary
- Snabb åtkomst av *värde* givet en viss nyckel
- Alla *värden* ska vara *unika* på samma sätt som nycklar i Dictionary är unika så att värje värde ger snabb åtkomst av motsvarande nyckel.

Vad som ska skilja mot ett vanligt Dictionary är alltså att man både får snabb åtkomst av värde givet en viss nyckel men också tvärt om. Klassen kommer alltså beskriva det matematiska begreppet bijektion. Vi utgår från Dictionary, men lägger till en instansvariabel som ska hålla ett vanligt Dictionary där värderna är nycklar och nycklarna värden. På det sättet använder vi vårt extra Dictionary för att snabbt plocka fram nyckel givet ett visst värde.

```
Dictionary variableSubclass: #BidirectionalDictionary
    instanceVariableNames: 'viceVersa '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Variabla-Strukturer'
```

Det behövs metoder för att läsa av och sätta den omvända lexikonet

```
viceVersa
    ^viceVersa

setViceVersa: aDict
    viceVersa := aDict          endast för  metoder som copyEmpty och postCopy
```

Initiering av det omvända lexikonet

```
initialize
    viceVersa := Dictionary new
```

För att koppla nyckel till värde används i första hand at:put:. Vi kan till stora delar bygga på superklassens at:put:, men vi måste också göra den omvända kopplingen för viceVersa. Där ska värdet anObject vara nyckel och argumentet key dess värde.

Då det kan ha funnits en koppling från anObject till en annan nyckel än key så måste vi först ta bort gamla kopplingar.

```
at: key put: anObject
    "Om nyckel inte hittas skapas en ny. Sätter värdet för nyckeln till anObject.
    Returnerar anObject."
    key isNil | anObject isNil                    felkoll innan något förändras
        ifTrue: [^self subscriptBoundsError: nil].
    self removeKey: (self viceVersa at: anObject ifAbsent: [])
        ifAbsent: [].
    self viceVersa at: anObject put: key.
    ^super at: key put: anObject
```

Vi skriver om metoden keyAtValue:ifAbsent: så att vi utnyttjar det omvända lexikonet för att öka snabbheten vid sökande efter nyckel givet ett värde. Eftersom metoden keyAtValue: använder sig av keyAtValue:ifAbsent: så behöver bara den senare skrivas om.

```
keyAtValue: value ifAbsent: exceptionBlock
    "svarar med den nyckeln vars värde är value. "
    "Om det inte finns någon sådan nyckel utförs exceptionBlock"
    ^self viceVersa at: value ifAbsent: exceptionBlock
```

När man skapar ett Dictionary så behöver man inte tänka på om objekten får plats. På låg nivå finns det alltid en maxstorlek för behållare. Det är den storleken som anges då man skapar objektet med new: storlek. En del av behållarklasserna döljer emellertid att det finns en maxstorlek genom att automatiskt öka storleken om behållaren blir full. Det skapas då ett nytt objekt med större storlek än originalet, därefter läggs alla objekt i originalobjektet in i det nya objektet och till sist sätts alla pekare om som pekar på det gamla objektet att peka på det nya och tvärt om. Detta sker genom att anropa metoden become:.

Normalt behöver man inte tänka på det, men när man definierar en subklass till en klass som ändrar storlek automatiskt så kan man

behöva ta hänsyn till det. Då det nya objektet skapas anropas metoden copyEmpty: newSize som ska skapa en kopia av mottagaren men *utan några element i vektordelen* av objektet. Däremot ska instansvariabler kopieras för att även det nya originalet ska ha kvar dem. I vårt fall vill vi att det nya objektet också ska ha en kopia till det omvända lexikonet.

```
copyEmpty: aSize
    "Returnerar en kopia av mottagaren som inte innehåller några element"
    ^(self class new: aSize) setViceVersa: self viceVersa
```

Som vi sett i kapitel 3 så fungerar metoden copy så att endast objektet och inte eventuella instansvariabler kopieras. Det betyder att vi måste kopiera instansvariabeln viceVersa så att kopian inte delar originalets viceVersa lexikon. Detta gör vi genom att definiera om metoden postCopy, som anropas av copy.

```
postCopy
    "Kopierar det omvända lexikonet också eftersom det ska vara
    unikt för varje instans av mig"
    super postCopy.
    self setViceVersa: self viceVersa copy
```

För att lägga till objekt i ett lexikon kan add: användas, men då krävs att argumentet är en instans av Association. Dess nyckel blir nyckel i lexikonet och värdet blir associerat med nyckel. I vårt fall ska också värdet användas som nyckel i det omvända lexikonet och där värdet istället blir associationens nyckel.

```
add: anAssociation
    "Lägger till anAssociation som ett av mina element"
    anAssociation key == nil ifTrue: [^self subscriptBoundsError: nil].
    anAssociation value == nil ifTrue: [^self subscriptBoundsError: nil].
    self removeKey: (self viceVersa at: anAssociation value ifAbsent: [])
        ifAbsent: [].
    self viceVersa at: anAssociation value put: anAssociation key.
    ^super add: anAssociation
```

Metoden removeKey:ifAbsent: behöver skrivas om så att associationen tas bort även ur det omvända lexikonet.

```
removeKey: key ifAbsent: aBlock
    "Tar bort key och det värde som finns associerat med nyckel. Om
    nyckeln inte finns svara med aBlock annars svara med det
    värde som finns associerat med nyckeln"
    | value |
    value := super removeKey: key ifAbsent: [^aBlock value].
    self viceVersa removeKey: value.
    ^value
```

Det är också möjligt att optimera metoden includes: genom att slå upp argumentet i det omvända lexikonet istället för att iterera över alla lexikonets värden och se om något är lika med argumentet.

```
includes: anObject
    "Returnerar true om anObjectingår i detta lexikon"
    self viceVersa at: anObject ifAbsent: [^false].
    ^true
```

Metoden occurencesOf: kan också skrivas om så att includes: används eftersom ett objekt inte kan läggas in som värde för mer än en nyckel.

```
occurrencesOf: anObject
    "Returnerar antalet element som är lika som anObject."
    ^(self includes: anObject)
        ifTrue: [1]
        ifFalse: [0]
```

Metoden new: på klassidan behöver skrivas om så att initialize anropas.

```
new: anInteger
    "Skapar ett nytt BidirectionalDictionary"
    ^(super  new: anInteger) initialize
```

Nu kan vi se om BidirectionalDictionary fungerar som avsett.

```
| dict |
dict := BidirectionalDictionary new: 1.
dict at: 'hej' put: 'hello'.
dict at: 'hund' put: 'dog'.
Transcript show: dict printString ; cr.
```

⇒*BidirectionalDictionary ('hej'->'hello' 'hund'->'dog')*

```
dict add: 'tja' ->'hello'.
Transcript show: dict printString ; cr. "
```

⇒*BidirectionalDictionary ('hund'->'dog' 'tja'->'hello')*

Nyckeln 'hej' har försvunnit eftersom vi endast tillåter att värdet är kopplat till en nyckel i taget.

```
dict viceVersa .
```

⇒*Dictionary ('dog'->'hund' 'hello'->'tja')*

Det omvända lexikonet har associationer åt motsatt håll.

```
dict keyAtValue: 'dog'.
```

⇒*'hund'*

Uppslagningen från värdet 'dog' till nyckel 'hund' är riktig.

```
dict occurrencesOf: 'hund'
```

⇒*0*

Strängen 'hund' förekommer inte som värde så noll är rätt svar.

8.8 Exempel

Person

För att som i Name i kapitel 4 också tvinga fram att en Person, från samma kapitel, endast kan skapas med både för- och efternamn givet gör vi några förändringar. Samtidigt så lägger vi till ett attribut identification. Detta attribut är tänkt att vara unikt per instans av Person.

```
Object subclass: #Person
    instanceVariableNames: 'name address identification '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Personer'
```

KLASSMETODER

```
new
    self shouldNotImplement

christianNames: aCollection familyName: aString identification: uniqueID
    ^self basicNew initialize
        christianNames: aCollection
        familyName: aString
        identification: uniqueID

christianNames: aChristianNames1 familyName: aFamilyName2
identification: uniqueID
    name := Name christianNames: aChristianNames1
                familyName: aFamilyName2.
    identification := uniqueID      Finns ingen åtkomstmetod som sätter om detta
                                    attribut så vi får göra en tilldelning
```

INSTANSMETODER

```
initialize
    address := Address new mm  m    mm
```

Med denna definition av Person är testet isValid, från sidan 91, överflödigt då alla instanser ska vara definierade med värden på dessa. Så nu kan vi ta bort dessa metoder!

För att få ett trevligare gränssnitt mot Person definierar vi också följande metod.

```
street: aStreetString number: aNumber city: aCity country: aCountry zip:
aZip phone: aPhone
    self address
    street: aStreetString
    number: aNumber
    city: aCity
    country: aCountry
    zip: aZip
    phone: aPhone
```

För att ytterligare förenkla gränssnittet och testning gör vi också följande förändringar: Ta bort Person>>initialize och Person>>christianNames:familyName:identification:.

Skriv instansmetoden

```
christianNames: aChristianNames1 familyName: aFamilyName2
identification: uniqueID address: anAddress
    name := Name christianNames: aChristianNames1
                familyName: aFamilyName2.
    address := anAddress.
    identification := uniqueID
```

och klassmetoden

```
christianNames: aCollection familyName: aString identification: uniqueID
address: anAddress
    ^self basicNew
        christianNames: aCollection
        familyName: aString
        identification: uniqueID
        address: anAddress
```

Samt ändra klassmetoden

```
christianNames: aCollection familyName: aString identification: uniqueID
    ^self
        christianNames: aCollection
        familyName: aString
        identification: uniqueID
        address: Address new
```

Med dessa definitioner och några ”systemklasser” kan vi nu konstruera ett enkelt personregister. Först i form av ett test i ett arbetsfönster.

```
| register p1 p2 p3 p4 nadaAddress |
register := OrderedCollection new.
p1 := Person
        christianNames: #(Olle John)
        familyName: 'Dahl' identification: 1.
p1 address country: 'Norge'.
p1 address city: 'Oslo'.
nadaAddress := Address
                    street: 'Osquars backe' number: 2
                    city: 'Stockholm'
                    country: 'Sverige'
                    zip: '10044'
                    phone: '08-7906000'.
register add: p1.
p2 := Person
        christianNames: #(Björn Arne)
        familyName: 'Eiderbäck'
        identification: 2
        address: nadaAddress.
```

```
p3 := Person
        christianNames: #(Olle)
        familyName: 'Bälter'
        identification: 3
        address: nadaAddress.
p4 := Person
        christianNames: #(Per)
        familyName: 'Hägglund'
        identification: 4
        address: nadaAddress.
register add: p2.
register add: p3.
register add: p4.
```

Nu kan vi använda registret för att göra vissa operationer och ställa vissa frågor.

Om vi tex vill skriva ut förnamnet på alla personer som bor i Sverige.

```
register
    do: [:aPerson | aPerson address country = 'Sverige'
            ifTrue: [Transcript cr.
                    Transcript print: aPerson christianName.
                    Transcript endEntry]]
```

Personregister

Nu konstruerar vi ett enkelt personregister. Vi väljer att inte kommentera det hela utan lämnar till läsaren att själv sätta sig in i klassen. Men detta skall säkerligen inte vara något problem.

```
Object subclass: #Register
    instanceVariableNames: 'persons '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Personregister'
```

INSTANSMETODER

initialize-release

initialize
```
    self persons: Dictionary new
```

accessing

persons
```
    ^persons
```

persons: newValue
```
    persons := newValue
```

enumerating

collect: aBlock
```
    ^self persons collect: aBlock
```

```
detect: aBlock
    ^self persons detect: aBlock
do: aBlock
    self persons do: aBlock
inject: thisValue into: binaryBlock
    ^self persons inject: thisValue into: binaryBlock
reject: aBlock
    ^self persons reject: aBlock
select: aBlock
    ^self persons select: aBlock
```

dictionary removing

```
removePersonWithID: id
    ^self removePersonWithID: id
                    ifAbsent: [self error: 'Ingen person med id: ' , id printString ,
    ' finns i registret!!']
removePersonWithID: id ifAbsent: aBlock
    ^self persons removeKey: id ifAbsent: aBlock
```

dictionary testing

```
includesIdentification: id
    ^self persons includesKey: id! !
```

adding

```
add: aPerson
    self persons at: aPerson identification put: aPerson!
addAll: aCollectionOfPersons
    aCollectionOfPersons do: [:each | self add: each].
    ^aCollectionOfPersons
```

KLASSMETODER

instance creation

```
new
    self shouldNotImplement
withAll: aCollectionOfPersons
    ^self basicNew initialize addAll: aCollectionOfPersons; yourself
```

Vi prövar registerklassen.

a) Vi utnyttjar en OrderedCollection och lägger in några fiktiva instanser av Person.

```
| personCollection reg |
personCollection := OrderedCollection new.
1 to: 10 do: [:i | | person chNames |
        chNames := OrderedCollection new: i.
        i to: 1 by: -1 do: [:j | chNames add: (String new: j
                                        withAll: (Character digitValue: j))].
        person := Person
```

```
            christianNames: chNames
            familyName: 'Family' , i printString
            identification: i.
            person address
            street: 'Gatan'
            number: i
            city: 'Stockholm'
            country: 'Sverige'
            zip: '10000'
            phone: '08-00123456789'.
        i \\ 3 = 0
            ifTrue: [person addressWork
                        street: 'Osquars backe'
                        number: 2
                        city: 'Stockholm'
                        country: 'Sverige'
                        zip: '10044'
                        phone: '08-7906000'
                        email: i printString , '@nada.kth.se'
                        fax: '08-7900930'].
    personCollection add: person].
```

b) Nu skapar vi ett register med alla personer i listan.

```
reg := Register withAll: personCollection.
```

c) För att vi ska kunna fråga en person om en adress till arbetet finns definierad så skriver vi också följande metod i Person.

```
hasAddressWork
    ^addressWork notNil
```

Nu kan vi välja alla personer som har en arbetsadress och lista deras identifikationer och email.

```
(reg select: [:p | p hasAddressWork])
    collect: [:p | 'ID: ', p identification printString, ' Email: ', p addressWork email]
```

med resultatet:

```
OrderedCollection (  'ID: 9 Email: 9@nada.kth.se'
                     'ID: 3 Email: 3@nada.kth.se'
                     'ID: 6 Email: 6@nada.kth.se' )
```

Sammanfattning

Termer

Länkad lista behållare som använder pekare till elementen.

Variabel subklass klass med indexerbara variabler. Finns två olika typer nämligen:

- med variabler av pekartyp
- med variabler av icke pekartyp där elementen är bytes

Metodik

Isolera implementationsspecifika delar till avgränsade partier av koden genom att lägga dem i egna metoder.

Definiera metoder utgående från andra metoder. Exempelvis kan man definiera massor av styrstrukturer så fort som do: fungerar för en klass.

Använd abstrakta superklasser för att definiera gemensamt gränssnitt för besläktade klasser.

Programeringsstil

Skriv fungerande kod först, genom att bla pröva den i arbetsfönster. Optimera först när allt fungerar.

Använd om möjligt befintliga behållare och skriv endast egna om det är absolut nödvändigt.

Oftast är en teknik med aggregering att föredra framför subklassning.

Smalltalk

copyEmpty och postCopy behöver ibland omdefinieras då man subklassar behållarklasser.

Man kan själv skapa klasser av variabel pekar- eller bytetyp.

I en klass av variabel pekartyp blir alla element nil vid instansiering och i en av bytetyp får alla element värdet 0 (noll) från början.
Tex

```
Array new: 3
⇒#(nil nil nil)
ByteArray new: 3
⇒#[0 0 0].
```

Övningar

8.1 skriv metoden do: i Vector som definierades på sidan 292 med hjälp av whileTrue:

8.2 Utgå från definitionen av begreppet likhet från sidan 295. Ge en svagare definition där vi släpper på att species ska vara samma, dvs att behållarna ska vara av samma klass.

8.3 Skriv om removeLast sidan 279 som rekursiv metod.

8.4 Pss som SimpleStack på sidan 280, implementera en kö.

8.5 Implementera en dubbelt länkad cirkulär lista med huvud.

8.6 Skriv styrstrukterer till den cirkulära listan i föregående övning.

8.7 Skriv en klass som hanterar köer av sköldpaddor. Modifiera sköldpaddorna om så krävs.

8.8 Skriv en klass som hanterar listor med två ändar, man skall kunna lägga till i slutet och början men bara kunna ta bort i början. Det skall vara möjligt att iterera över elementen i listan.

8.9 Använd listan från övning 8.8 för att lösa övning 7.3. Detta kan göras på flera sätt, prova.

9 Beroenden

I det här kapitlet besvaras bland annat följande frågor:

- Hur kan objekt och deras förändringar göras så oberoende av omgivningen som möjligt?
- Hur kan objekt meddela sig med sin omgivning utan att göra antaganden om den?
- Hur hanterar Smalltalk objekt på ett omgivningsoberoende sätt?
- Hur används update- och changed-meddelanden?
- Vad är en adapter?

En viktig egenskap hos objektorienterade språk är att de gör det möjligt att dölja den inre strukturen med hjälp av inkapsling. Viktigt är också att objekt strävar efter att vara oberoende av sin omgivning. Strävan är att definiera objekt så att de ska fungera i olika sammanhang utan att behöva förändras.

Vi kommer att utgå från ett personregister med personer sorterade i bokstavsordning med avseende på sina efternamn. Vi kommer diskutera tekniker för att hantera att personer ändrar efternamn och hur registret då kan göras medvetet om detta för att sortera in personerna på rätt plats. Vi kommer basera våra lösningar på tekniker baserade på *beroenden* (eng dependents) mellan personer och register. Vi ska peka på några fördelar med denna teknik, bla att de olika objekten blir ”löst kopplade” till varandra och därmed inte så känsliga för smärre förändringar i beskrivningen av de andra.

Beroenden är speciellt användbara vid konstruktion av interaktiva grafiska tillämpningar. Speciellt användbar är tekniken om objekten ska ha flera olika samtidiga presentationssätt eller om olika sätt att interagera med dem ska erbjudas. Speciellt tydligt blir detta i en teknik som brukar kallas för *Model View Controller* (MVC), se referenser sidan 455 [17] och [19].

I MVC delas en grafisk interaktiv tillämpning på tre olika delar:

- Model

  Hanterar datastrukturer, modeller och mer tillämpningsorienterat beteende

- View

  Definierar hur objekten ska presenteras på skärmen

- Controller

  Beskriver hur objekten ska interagera med användarna

Förbindelserna mellan den grafiska och interaktiva delen och modellen hanteras med beroenden. Denna lösa koppling möjliggör att flera olika vyer enkelt kan konstrueras på samma modell. Om modellen förändras kan den enkelt, utan antaganden om presentationsdetaljer, meddela detta till presentationsobjekten.

9.1 Enkel personlista

För att enkelt kunna illustrera beroenden med ett lite mer praktiskt inriktat problem konstruerar vi nu en enkel klass för att hantera ett register av personer. Registret använder en SortedCollection för att hela tiden hålla registret sorterat i bokstavsordning med avseende på personernas efternamn. Vi antar att personerna som läggs in i registret returnerar sina respektive efternamn om meddelandet familyName skickas till dem.

Vi kallar klassen för PersonList och implementerar den enligt följande:

```
Object subclass: #PersonList
    instanceVariableNames: 'persons '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Personregister-enkelt'

INSTANSMETODER

initialize-release

initialize
    persons := SortedCollection
                    sortBlock: [:personOne :personTwo |
                        personOne familyName <= personTwo familyName]

accessing

persons
    ^persons
```


adding

add: aPerson
"Stoppa in en ny person i registret"
^self persons add: aPerson

removing

remove: aPerson
"Ta bort personen ur registret. Om personen inte finns händer ingenting."
^self persons remove: aPerson ifAbsent: []

KLASSMETODER

instance creation

new
^super new initialize

9.2 Personlista och personer med beroenden

Om en person i en instans av PersonList skulle byta efternamn och därmed hamna på en annan plats i bokstavsordningen skulle detta inte få någon effekt på listans sortering. Om det ska får effekt måste listan speciellt uppmanas att sortera om sig (meddelandet reSort). Detta är inte riktigt effektivt utan nu ska vi beskriva hur listan kan hållas sorterad även om personer ändrar namn.

Ett sätt är att implementera en speciell mutator i registret som alltid används för eventuella ändringar av personernas namn. I så fall kan vi enkelt se till att registret också sorterar in personen på rätt plats vid en eventuell namnförändring. Men i vissa sammanhang är denna lösning opraktisk. Om tex en viss person finns med i många olika personregister, så måste vi för att personen ska bli insorterad på rätt plats ändra personens namn i varje register. Vi ska visa att det ofta är bättre om personobjektet själv är ansvarig för att meddela intresserade objekt vid en förändring.

Vi konstruerar ett enkelt personobjekt. För att personobjektet ska kunna hålla reda på vilka register det finns i så lägger vi till instansvariabeln dependents. Instansvariabeln är tänkt att innehålla alla objekt som är beroende av den aktuella instansen.

```
Object subclass: #Person
    instanceVariableNames: 'dependents name address '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Personer-med-beroenden'
```

Vi initierar dependents så att den till en början innehåller en tom mängd. Vi skriver också några metoder som behövs för att skapa nya instanser samt för att sortera in dem i listan. Klassen Name är ett enkelt objekt som ska hantera en persons för- och efternamn och kan tex hämtas direkt från avsnitt 3.13.

INSTANSMETODER

```
initialize
    "tilldela dependents en tom mängd"
    dependents := Set new

familyName
    "Returnera efternamnet (behövs i PersonList)"
    ^self name familyName

christianNames: aChristianNames familyName: aFamilyName
    "Konstruerar en instans av Name och tilldelar den namnuppgifterna"
    name := Name christianNames: aChristianNames
                familyName: aFamilyName
```

KLASSMETODER

```
christianNames: aCollection familyName: aString
    "Konstruerar en ny instans givet förnamnen i en lista och eftenamnet av en
    sträng"
    ^super new
        christianNames: aCollection
        familyName: aString
```

Vi behöver nu metoder för att till dependents lägga till och ta bort referenser till objekt som är intresserade att veta om ett Person-objekt har förändrats.

```
addDependent: anObject
    "lägg anObject till beroendelistan. Returnera anObject."
    dependents isNil ifTrue: [dependents := Set new].
    ^dependents add: anObject

removeDependent: anObject
    "ta bort anObject från beroendelistan. Returnera anObject"
    ^dependents remove: anObject ifAbsent: [anObject]
```

Vi har nu möjlighet att både lägga till och ta bort objekt ur listan av objekt som är intresserade av förändringar.

För att personlistor automatiskt ska bli beroende av nya personer som läggs in i dem utökar vi metoden PersonList>>add: så listan läggs till den nya personens beroendelista.

```
add: aPerson
    "Gör mig beroende av aPerson och stoppa in personen i personlistan"
    aPerson addDependent: self.
    ^self persons add: aPerson
```

När en person tas bort från registret måste vi också se till att registret tas bort från personens beroendelista. Så vi ändrar också PersonList>>remove:.

```
remove: aPerson
    "Ta bort mitt beroende av aPerson och plocka bort aPerson från personlistan"
    aPerson removeDependent: self.
    ^self persons remove: aPerson
```

Nu kan vi definiera om meddelandet lastName: i Person så att alla beroenden får veta att efternamnet har ändrats.

```
lastName: newLastName
    self name lastName: newLastName.
    self dependents do: [:aDependent | aDependent lastNameChangedFor: self]
```

Nu måste vi i PersonList implementera metoden lastNameChangedFor: så att listan sorteras om någon ändrar efternamn.

```
lastNameChangedFor: aPerson
    "Det enklaste sättet att sortera in personen på rätt ställe är att först ta bort
    personen och sedan stoppa in den igen."
    self persons remove: aPerson.
    self persons add: aPerson.
```

För att få aPerson på rätt plats i registret så tar vi helt enkelt först bort aPerson, och därefter lägger till den igen. Den här gången ser listan till att personobjektet hamnar på rätt plats.

Om vi senare vill införa en ny form av lista tex sortering med avseende på bostadsadress så måste vi skriva om metoden address: på samma sätt:

```
address: newAddress
    address := newAddress.
    self dependents do: [:aDependent | aDependent adressChangedFor: self]
```

Naturligtvis måste vi i Addresslist också implementera metoden adressChangedFor: enligt

```
addressChangedFor: aPerson
    self persons remove: aPerson.
    self persons add: aPerson.
```

Men det räcker inte med detta. Även PersonList måste implementera addressChangedFor:. Om adressen ändras så kommer förstås meddelandet skickas till alla objekt som har gjort sig beroende av den aktuella personen.

Även AddressList måste på samma sätt implementera metoden lastNameChangedFor:. Detta gör att så fort vi behöver få reda på om någon ytterligare egenskap av objektet har ändrats så måste vi implementera nya metoder för alla som har gjort sig beroende av objektet. Det betyder

att objektet Person gör antaganden om i vilken omgivning den används och vilka meddelanden som den förstår och vi har därmed skrivit en klass som är svår att använda i nya sammanhang. Vi bryter också mot önskemålet att delegera uppgifter till den det vederbör, eftersom Person måste känna till hur den används och i vilka register den finns. I nästa avsnitt visar vi hur vi kan undvika detta dilemma.

9.3 Meddelandet update

Ett sätt att komma runt problemet med att Person gör antaganden om hur den används är att använda samma meddelande oavsett hur objektet har förändrats. I stället används en parameter som beskriver vad som har förändrats. Vi inför därför en metod update: aspectSymbol with: aParameter from: sender där aspectSymbol beskriver vilken förändring som har skett, aParameter anger hur och sender anger vilket objekt som har ändrats. I personregistret definierar vi två olika egenskaper #lastName och #address som anger om personobjektets namn eller address har förändrats.

Med denna lösning får vi skriva om metoderna lastName: och adress:.

```
lastName: newLastName
    self name familyName: newLastName.
    self dependents do: [:aDependent |
            aDependent update: #lastName with: newLastName sender: self]

address: newAddress
    address := newAddress.
    self dependents do: [:aDependent |
            aDependent update: #address with: newAddress sender: self]
```

Metoderna addressChangedFor: och lastNameChangedFor: ersätts med metoden update:with:from: som implementeras i alla klasser vars instanser ska kunna vara beroende av personobjekt. Implementeringen i PersonList blir:

```
update: aspectSymbol with: aParameter from: sender
    aspectSymbol == #lastName                    endast efternamn är intressant
        ifTrue: [self persons remove: sender.
                self persons add: sender]
```

I AddressList blir update-metoden analog

```
update: aspectSymbol with aParameter from: sender
    aspectSymbol == #address                     endast adressen är intressant
        ifTrue: [self persons remove: sender.
                self persons add: sender]
```

9.4 Meddelandet changed

För att underlätta användandet av beroenden döljer vi itererandet över alla som har gjort sig beroende av objektets förändringar i en speciell metod. Vi implementerar en metod changed: with: som itererar över listan av beroende objekt och skickar update-meddelanden till dem.

I Person ser det hela ut som följer:

```
changed: anAspectSymbol with: aParameter
    self dependents
        do: [:aDependent |
                aDependent update: anAspectSymbol
                        with: aParameter
                        sender: self]
```

I personexemplet använder vi oss inte av with:-parametern utan den är alltid nil. Vidare vore det trevligt att inte behöva skriva ut hela meddelandet changed:with: om inte argumentet till nyckelordet with: används, utan bara skicka med parametern aspectSymbol. Detta kan vi enkelt åstadkomma genom att införa metoden

```
changed: anAspectSymbol
    self changed: anAspectSymbol with: nil
```

På motsvarande sätt går det att förkorta meddelanden med många parametrar, där en eller flera av parametrarna ofta är desamma från anrop till anrop. Man får alltså en form av skönsvärden för de parametrar som inte särskilt anges.

Metoderna lastName: och address: blir nu

```
lastName: newLastName
    self name familyName: newLastName.
    self changed: #lastName

address: newAddress
    address := newAddress.
    self changed: #address
```

Om vi summerar så har vi gjort följande förändringar i klassen Person:

- Instansvariabeln dependents har lagts till för att hålla reda på de objekt som vill få reda på när ett personobjekt har ändrats.
- Meddelandena changed: och changed:with: har lagts till för att vi på ett enkelt sätt ska kunna meddela att efternamn eller adress har ändrats.
- Metoderna lastName: och address: har definierats om och anropar nu changed-meddelandet så att de olika listorna får veta att en person har bytt efternamn eller adress.

De första två punkterna ovan är inte på något sätt beroende av vår implementation av klassen Person utan dessa kan flyttas till en superklass och därmed användas av andra klasser. Vi kan alltså införa en ny superklass SimpleModel enligt

```
Object subclass: #SimpleModel
    instanceVariableNames: 'dependents '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'My-Kernel-Objects'
```

Den ska också förstå meddelandena changed:, changed:with: och skicka update-meddelanden till beroende objekt på samma sätt som Person gjorde förut. Det behövs även en del metoder för initiering och att frigöra minne.

```
INSTANSMETODER

initialize-release

initialize
    "initierar dependents för mig"
    dependents := Set new

release
    "tar bort alla referenser till objekt som gjort sig beroende av mottagaren"
    self breakDependents

dependents access

addDependent: anObject
    "lägger till anObject som en av mina beroenden"
    self dependents add: anObject.
    ^anObject

dependents
    ^dependents

removeDependent: anObject
    "tar bort argumentet från mängden av mina beroenden"
    self dependents remove: anObject ifAbsent: [].
    ^anObject

changing

changed
    "mottagaren har ändrats informera alla som är beroende"
    self changed: nil

changed: anAspectSymbol
    "Mottagaren har ändrats, informera alla som är beroende. Typen av
    förändring beskrivs av anAspectSymbol."
    self changed: anAspectSymbol with: nil
```

```
changed: anAspectSymbol with: aParameter
    "Mottagaren har ändrats, informera alla som är beroende. Typen av
    förändring beskrivs av anAspectSymbol. aParameter ger ofta ytterligare
    information om förändringen "
    self dependents do: [: aDependent |
            aDependent update: anAspectSymbol with: aParameter from: self]
```

private

```
breakDependents
    "tar bort mina referenser till dem som gjort sig beroende av mottagaren"
    dependents := Set new
```

KLASSMETODER

instance creation

```
new
    "skapa en ny instans och initiera den"
    ^super new initialize
```

9.5 Systemets lösning

Det här sättet med referenser mellan objekt som är beroende av varandra används flitigt av Smalltalksystemet självt. Framförallt för att informera objekt som syns på skärmen om förändringar hos underliggande objekt. Därför har metoder för att hantera beroenden placerats högst upp i klasshierarkin och är implementerade i klassen Object.

I klassen Object finns metoderna changed: och changed:with: definierade tillsammans med addDependent: och removeDependent: för att lägga till respektive ta bort objekt ur listan med beroende objekt.

Klassen Model som finns i systemet hanterar beroenden på ett effektivt sätt. Den liknar i mycket klassen SimpleModel som beskrevs i föregående avsnitt. Alla klasser i systemet som bygger på att de skickar changed-meddelanden är av effektivitetsskäl subklasser till klassen Model.

För att alla objekt ska kunna göra sig beroende av andra objekt så har man också implementerat meddelandet update:with:from: i Object. Meningen är att meddelandet ska definieras om i subklasser som vill beskriva vad som ska hända om ett objekt de är beroende av förändras.

Implementationen av update:with:from: är lite speciell eftersom den skalar bort parametern from: och skickar det enklare meddelandet update:with: till objektet själv. Detta för att den som inte är intresserad av sändaren ska kunna definiera om metoden där sändaren inte måste vara en parameter. Definitionen är

```
update: anAspectSymbol with: aParameter from: aSender
    ^self update: anAspectSymbol with: aParameter
```

På motsvarande sätt är update:with: implementerad dvs den skalar bort en parameter och skickar det nya meddelandet.

```
update: anAspectSymbol with: aParameter
    ^self update: anAspectSymbol
```

Meddelande update: är det sista i hierarkin och definitionen är

```
update: anAspectSymbol
    ^self
```

Dvs update: gör egentligen ingenting.

Ett objekt kan alltså göras beroende av vilket objekt som helst i systemet och förutsatt att detta objekt skickar changed-meddelanden så får beroende objektet reda på förändringar. Hur är det då möjligt för ett objekt som inte är subklass till Model att hålla reda på de objekt som är beroende av det? Det har ju inga direkta referenser till objekten.

Svaret är att det finns en variabel som håller i alla beroenden i systemet utom till de objekt som är subklass till Model. Denna variabel finns definierad som en klassvariabel till Object och heter DependentsField. Inspektionsfönstret kommer visa ett Dictionary där nycklarna är de objekt som någon är beroende av och deras värden är de beroende objekten.

Spara före avslut

Om personregistret förändras vore det bra om användaren blev påmind om att spara registret på disk innan Smalltalk avslutas.

Systemet har förberett för detta genom att det objekt som hanterar start, avslut, samt sparar omgivningen informerar alla intresserade med hjälp av changed-meddelanden när så sker. Objektet som hanterar detta är klassen ObjectMemory. Vid avslut och start av imagen skickas changed-meddelanden till de objekt som är beroende av den med bla symbolerna:

- #aboutToSnapshot — precis innan imagen sparas
- #returnFromSnapshot — efter att imagen sparats, den kan antingen bara ha sparats eller varit avstängd och startats på nytt.
- #aboutToQuit — innan systemet avslutas

I vårt fall är det aboutToQuit som är intressant eftersom vi vill veta när systemet håller på att stoppas och först ge användaren av adressregistret en möjlighet att spara det. För att få det att fungera måste personre-

gistret registreras som beroende av ObjectMemory tex genom att utöka initialize.

```
initialize
    persons := SortedCollection
                    sortBlock: [:personOne :personTwo |
                        personOne familyName <= personTwo familyName].
    ObjectMemory addDependent: self.
```

Därefter implementerar vi en egen metod update:.

```
update: aSymbol
    (aSymbol == #aboutToQuit
            and: [(Dialog
                    choose: 'Vill du spara adressregistret?'
                    labels: #( 'Ja' 'Nej')
                    values: #(#ja #nej)
                    default: #ja) == #ja])
        ifTrue: [self save]
```

En varning angående potentiella fel i koden om ett objekt är beroende av ObjectMemory är på sin plats. Eftersom ObjectMemory skickar changed-meddelanden, bla då systemet startar, så är det viktigt att det inte är fel i update-metoden eftersom systemet inte har startat upp helt och hållet då den utförs. Om det skulle vara fel i den så kan det leda till att systemet kraschar och imagen inte längre går att använda. Det är alltså viktigt att vara försiktig då objekt görs beroende av ObjectMemory.

Meddelandet release

Vi har nu en fungerande lösning som frågar användaren om adressregistret ska sparas innan systemet stängs av. Men hur ska vi göra om vi inte längre vill använda ett visst register? Då måste vi se till att registret inte längre är beroende av ObjectMemory. Det går naturligtvis att skicka meddelandet ObjectMemory removeDependent: self så fort det inte längre används, men ett bättre sätt är att utöka metoden release som finns definierad i Object.

```
release
    ObjectMemory removeDependent: self.
    super release                                   anropa release i Object
```

När ett register inte längre ska användas så skickas meddelandet release till det och får därefter inte längre update-meddelanden från ObjectMemory.

Det är viktigt att tänka på att skicka meddelandet release till objekt som andra objekt kan vara beroende av då de inte längre används. Annars kan skräpsamlaren få problem att städa. Vilket i sin tur leder till att

objekt som inte längre används blir kvar och imagens storlek kommer att öka i onödan.

Några andra metoder som bygger på beroenden

- broadcast: aMessageSymbol skickar meddelandet aMessageSymbol till alla som är beroende av objektet
- changeRequest: aSymbol sänder meddelandet updateRequest: aSymbol till alla beroenden och dessa ska svara med true eller false beroende på om de tillåter uppdatering.

9.6 ValueModel

ValueModel med subklasser är klasser som baserar sin funktion på beroenden. De kan utgöra byggstenar för tillämpningar som behöver beroendefunktionalitet och underlättar för programmeraren att bygga modulära program. Grunden för alla dessa klasser är att de har ett uniformt yttre gränssnitt. I huvudsak förstår de meddelanderna value och value:. Då meddelandet value skickas till ett objekt, ur någon av ValueModels subklasser, så svarar objektet med ett värde. Meddelandet value:, som de flesta av subklasserna förstår, sätter om värdet.

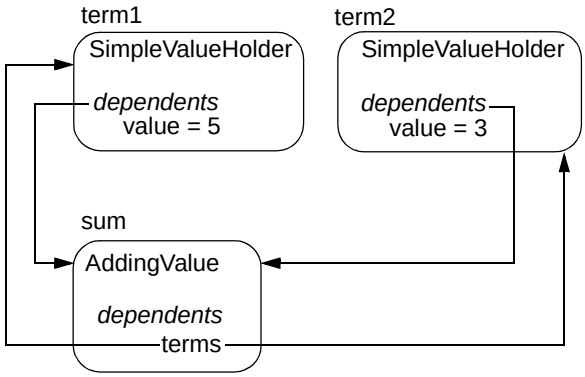
Då värdet ändras i en ValueModel så skickas ett changed-meddelande. Därför är dessa klasser lämpliga att använda om flera olika programdelelar ska dela på värden.

Genom att dessa klasser har ett enhetligt gränssnitt så behöver klienter till objekten inte exakt veta vilken av alla ValueModel-klasser som de hanterar eftersom alla har samma gränssnitt.

Följande scenario beskriver en situation då ValueModel med subklasser är användbara:

Vi vill beskriva ett system i form av en enkel adderare enligtfigur 9.1

Vi har här två objekt som representerar termer i summan och ett objekt som representerar summan. Vi vill kunna ändra värdet av termerna. Då termerna ändras ska summan också ändras. Det ska alltså fungera ungefär som ett kalkyprogram där man kan ändra i celler därefter ändras automatiskt alla celler som är beroende av den ändrade cellen, direkt eller indirekt. En möjlighet att åstadkomma detta är att utnytja två av ValueModel's subklasser. Först behöver vi en typ av objekt som kan lagra termerna och som informerar andra objekt när dess värde har ändrats. För att beräkna summan behöver vi en typ av objekt som refererar sina båda termer och som märker då dessa ändras och som då



Figur 9.1 Adderare med två termer

själv informerar de som gjort sig beroende. Objektet ska också beräkna summan av termerna då man skickar meddelandet value till det.

SimpleValueHolder

En huvuduppgift för termerna är att informera andra då de har förändrats. Därför är det rimligt att de är subklasser till Model så att de effektivt kan hantera beroenden. Vi kallar klassen för SimpleValueHolder och ger den en instansvariabel för att hålla aktuellt värde.

```
Model subclass: #SimpleValueHolder
    instanceVariableNames: 'value '
    classVariableNames: ' '
    poolDictionaries: '  '
    category: 'Dependents-example-test'
```

Vi definierar en inspektor

```
value
    ^value
```

Om värdet av en instans av SimpleValueHolder ändras vill vi att alla beroenden ska få veta det och därför skickar vi ett changed-meddelande om detta inträffar.

```
value: newValue
    "sätter om mitt värde och informerar därefter alla beroenden om detta"
    value := newValue.
    self changed: #value.
```

Om värdet ändras så informeras alla beroenden om det genom att de får ta emot ett update-meddelande, på det sätt som beskrevs i avsnitt 9.3.

Systemets Lösning

I VisualWorks finns en hierarki av värdehanterande klasser med den abstrakta klassen ValueModel som gemensam superklass, se figur 9.2 där också instansvariablerna för respektive klass är inkluderade.

```
Object ()
  Model ('dependents')
    ValueModel ()
      ComputedValue ('cachedValue' 'eagerEvaluation')
        ...
      PluggableAdaptor ('model' 'getBlock' 'putBlock' 'updateBlock')
        ...
      ProtocolAdaptor ('subject' 'subjectSendsUpdates' 'subjectChannel'
                       'accessPath')
        ...
      RangeAdaptor ('subject' 'rangeStart' 'rangeStop' 'grid')
      ValueHolder ('value')
        ...
```

Figur 9.2 Utsnitt av ValueModel's hierarki

Klasserna har det gemensamt att dess instanser lagrar värden antingen direkt eller indirekt i andra objekt och att värdet avläses genom att meddelandet value skickas till objekten. De flesta klasser definierar också meddelandet value: som tilldelar ett nytt värde. När värdet förändras så skickas ett update-meddelande till alla som är beroende av det.

Den vanligaste av dessa klasser är ValueHolder som håller i ett värde som både kan läsas och ändras. Då värdet ändras skickas ett changed-meddelande.

Det finns också möjligheter att ange att ett visst meddelande ska skickas till ett beroende objekt. Objekt får då detta meddelande om det den är beroende av förändras, istället för ett update-meddelande som skickas om beroendet etableras med addDependent:. I många situationer är detta beroende bättre då objekten direkt kan beskriva vad som ska hända om en viss förändring sker. Meddelandet som etablerar ett sådant beroende heter onChangeSend:to:.

Ett exempel på detta är

```
|aVH|
aVH := 12 asValue.                          skapa en ValueHolder med värdet 12
aVH onChangeSend: #value to: [Transcript show: 'värdet har ändrats' ; cr].
(Delay forSeconds: 5) wait.                                    vänta 5 sekunder
aVH value: 14.
```

Genom att meddelandet asValue skickas skapas en värdebehållare (ValueHolder) på heltalet 12. Meddelandet asValue är definierat i Object och därför kan alla objekt enkelt kapslas in i en ValueHolder.

Därefter uppmanas aVH att skicka meddelandet value till blocket om dess innehåll ändras. Blocket skriver ut en sträng i utskriftsfönstret. Efter fem sekunder ändras värdet till 14, och blocket får ta emot meddelandet value vilket innebär att 'värdet har ändrats' skrivs i utskriftsfönstret.

Det är möjligt att ta reda på vilket objekt som har ändrats genom att etablera beroendet med en metod med två argument. I resultatet kommer sedan det första argumentet alltid vara nil för subklasser till ValueModel medan det andra kommer vara den ValueHolder som ändrats. Om vi använder argument i exemplet så får vi:

```
| aVH |
aVH := 12 asValue.                          skapa en ValueHolder med värdet 12
aVH onChangeSend: #value:value:
      to: [:argument :sender |
            Transcript show: 'värdet har ändrats till ', sender value printString; cr].
(Delay forSeconds: 5) wait.                 vänta 5 sekunder
aVH value: 14.
```

⇒'värdet har ändrats till 14'.

Om ett objekt deklarerar att en metod med flera argument ska användas vid förändring skickas argument och avsändare till det från värdebehållaren då dess värde ändras. I det här fallet är argumentet nil, då en ValueHolder inte skickar med några specialla argument vid en förändring, däremot skickas alltid den aktuella värdebehållaren med som ett andra argument (här sender).

Summeringsobjektet

För att lösa sista delen av problemet med summeringsobjektet så kan vi gå till väga på flera olika sätt. Vi visar två som båda bygger på beroenden. Det som lösningarna har gemensamt är att summeringsobjektet ska vara beroende av sina värden och att summeringsobjektet informerar sina beroenden om det får veta att någon av termerna ändrats.

Vi konstruerar en subklass till ValueModel

Den första lösningen för att beskriva ett summeringsobjekt bygger på att definiera en subklass till ValueModel

```
ValueModel subclass: #AddingValue
    instanceVariableNames: 'terms'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Example-Dependents'
```

INSTANSMETODER

```
initialize
    "Initiera termlistan"
    super initialize.
    terms := OrderedCollection new.

release
    "Jag vill inte längre vara beroende av några objekt"
    super release.
    self terms do: [:term | term removeDependent: self]
```

Metoden release behöver utökas så att summeringsobjektet inte blir kvar pga sina beroenden då det inte längre behövs.

```
terms
    ^terms

update: anAspect
    "om anAspect är #value så har min summa också ändrats"
    anAspect = #value ifTrue: [self changed: #value]
```

Så fort adderaren får veta att någon av termerna har ändrats så meddelar den att den själv har ändrats också.

```
addTerm: aTerm
    "Addera en ny term och gör mig själv beroende av den"
    self terms add: aTerm.
    aTerm addDependent: self.

removeTerm: aTerm
    "Jag vill inte längre veta om aTerm ändras"
    self terms remove: aTerm.
    aTerm removeDependent: self

value
    "beräkna summan av alla termer"
    ^self terms inject: 0 into: [:sum :aVH | aVH value + sum]

value: aValue
    "jag beräknar bara totalsumma och kan inte tilldelas värde"
    self shouldNotImplement.
```

Nu har vi kommit så långt att vi kan implementera summeringsstrukturen där varje term lagras i SimpleValueHolder och AddingValue beskriver summan av termerna

```
| vh1 vh1  adder|
vh1 := SimpleValueHolder new value: 3.
vh2 := SimpleValueHolder new value: 3.

adder := AddingValue new.
adder addTerm: vh1.
adder addTerm: vh2.

adder onChangeSend: #value:value: to: [:null :sender | Transcript show:
'summan är ' , sender value printString ; cr].

vh1 inspect.
vh2 inspect.
```

Om man exekverar koden ovan så kommer två inspektionsfönster öppnas på skärmen. Ändra värdet i ett av dem genom att i textdelen skriva

```
self value: 5
```

måla över och välj **do it** i kommandomenyn. Titta nu i utskriftsfönstret. Där står det att summan har ändrats till åtta. Förändringen i valueHoldern har alltså förts vidare till adderaren och till sist har blocket frågat efter adderarens nya värdet och skrivit ut det.

Implementering med BlockValue

I Smalltalk finns flera klasser som hanterar objekt där man kan läsa av värdet och där man får reda på om värdet ändras. Vi kan för att beskriva en summering använda en klass som heter BlockValue och som gör sig beroende av flera objekt och som sedan beräknar ett värde baserat på dem. Klassen BlockValue har också den fördelen, jämfört med vår AdderValue, att den sparar undan det värde den beräknar så att den bara beräknar värdet på nytt om någon komponent har ändrats.

Med BlockValue blir exemplet med adderare

```
| vh1 vh2 arg adder |
vh1 := 1 asValue.
vh2 := 2 asValue.
arg := Array with: vh1 with: vh2.
adder := BlockValue block: [:v1 :v2 | v1 + v2 ] arguments: arg.
adder
    onChangeSend: #value:value:
    to: [:null :sender | Transcript show: 'summan är ' ,
                                sender value printString ; cr].
vh1 value: 3.
```

⇒*summan är 5*

```
vh2 value: 4
```

⇒*summan är 7*

9.7 Adaptorer

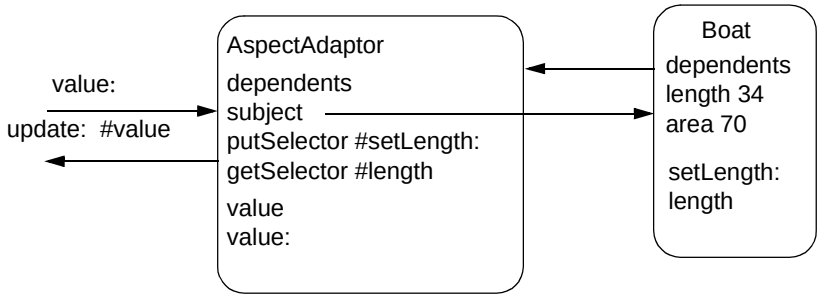
Adaptorer är ett slags kitt mellan programdelar som anpassar beteenden mellan olika objekt så att de kan kommunicera med varandra. Det kan i det enkaste fallet handla om att bara översätta meddelandenamn mellan objekt men det kan också handla om att göra olika former av transformationer på både argument och resultat vid metodanrop.

Man kan också tänka sig adaptorer som själva tillför funktionalitet på en generell nivå så att adaptorn blir återanvändbar. Utan återanvändbarheten blir det egentligen en utbyggnad av det ursprungliga objektet.

I detta avsnitt ska vi titta på några av de adaptorer som finns inbyggda i Smalltalk. Dessa adaptorers avsikt är att anpassa objekt till ValueModel så att de uppträder som en ValueModel, dvs förstår value, value: och skickar changed: #value då värdet förändrats. Vi ska titta närmare på AspectAdaptor som bara gör transformation på metodnamnen och PluggableAdaptor som är mer generell.

AspectAdaptor

AspectAdaptor är en enkel form av adapter som kopplas till ett objekt, som kallas subject. Med AspectAdaptor är det meningen att man ska komma åt en speciell egenskap hos subjektet. AspectAdaptor gör en enkel avbildning från meddelandet value till valfritt meddelande, m, och från value: till något annat meddelande. Om en AspectAdaptor får meddelandet update: #m så skickar den changed: #value.



Figur 9.3 Båt med adapter

För att koppla ihop en adapter med en Boat enligt figur 9.3 så gör man

```
| adaptor |
....
adaptor := AspectAdaptor subject: aBoat.
adaptor accessWith: #length assignWith: #setLength:.
....
```

För att hantera detta så använder en AspectAdaptor två instansvariabler: getSelector och putSelector. getSelector skickas till subjektet då adaptern får meddelandet value. putSelector skickas då adaptern får meddelandet value:.

PluggableAdaptor

Den enkla anpassningen till ValueHolders gränssnitt som aspectAdaptor gör räcker inte om man tex behöver konvertera värdet från en storhet till en annan. Då kan man använda den mer generella klassen PluggableAdaptor. Den kopplas mot ett subjekt och med hjälp av två block specificerar man vad som ska hända då meddelanderna value respektive value: skickas till adaptorn. Ett tredje block beskriver vad som ska hända då adaptorn mottar ett changed-meddelande från subjektet. Med en PluggableAdaptor kan man som exempel enkelt åstadkomma rationell konvertering av temperatur i Fahrenheit till Celsius och tvärt om.

```
| tempF tempC |
tempF := 100 asValue.
tempC := PluggableAdaptor on: tempF.
tempC getBlock:[:s | (s value -32) * 5/9]
        putBlock:[:s :newC | s value: 9/5 * newC +32]
        updateBlock:[:s :a :p | a==#value].
```

Vad betyder då allt detta?

- rad två tilldelar tempF en valueHolder med värdet 100.
- rad tre skapar en PluggableAdaptor vars subjekt är tempF, dvs som anpassar värdet av tempF.
- rad fyra till sex utgör en enda sats som beskriver hur konverteringen till och från Fahrenheit ska ske. I rad fyra beskriver vi hur man ska konvertera från Fahrenheit till Celsius, blocket får som argument subjektet dvs tempF, och utifrån det beräknas värdet adaptern ska returnera.
- rad fem konverterar från Celsius till Fahrenheit och denna gång har blocket två argument subjektet, precis som förut, samt den nya temperaturen som ska konverteras. I blocket beräknas sedan temperaturen i Fahrenheit som sedan tilldelas valueHoldern tempF, som är adapterns subjekt.

- rad sex beskrivar hur vår PluggableAdaptor ska skicka ett changed-meddelande. För att göra det behövs ett block med tre argument: subjekt, egenskap som har ändrats samt värde. Egenskapen och värdet motsvarar parametrarna till update: aspect with: value. Blocket ska returnera true om adaptern ska skicka ett change-meddelande och annars false.

Sammanfattning

Termer

Beroenden (eng. dependents) mellan objekt innebär att de kan deklarera intresse av andra objekts vitala förändringar. De andra objekten skickar sedan ut förändringsmeddelanden vid sådana förändringar.

Förändringsmeddelanden (changed-meddelanden) kan skickas till ett objekt när vital del har ändrats. Detta resulterar i att eventuella beroende objekt meddelas om förändringen genom att uppdateringsmeddelanden skickas till dem.

Uppdateringsmeddelanden (update-meddelanden) som resultat av att ett objekt skickar ett förändringsmeddelande får alla beroende objekt ett uppdateringsmeddelande som informerar om vad och vem som har förändrats.

Värdehållare (eng. value holder) ett objekt som håller i ett värde. Meddelar beroende objekt om värdet förändras.

Adapter ett objekt som anpassar gränssnitt mellan olika objekt.

Smalltalk

Metoderna changed, changed: och changed:with: kan anropas då objekt har ändrats.

Metoderna update:, update:with: och update:with:from: anropas då någon skickar changed-meddelanden

Metoderna addDependent: och removeDependent används för att lägga till/ta bort beroenden.

Model är en abstrakt klass som hanterar beroenden effektivt

ValueModel är en abstrakt klass för objekt som förstår value, value: och som skickar changed-meddelanden då de ändras

ValueHolder är den enklaste och vanligaste klassen i ValueModel-hierarkin och lagrar ett värde

ObjectMemory hanterar start, avslut och uppstart av imagen. Objekt kan göra sig beroende av denna klass och blir då via uppdateringsmeddedelanden informerade om dessa aktiviteter.

Övningar

9.1 Nu ska vi göra om sköldpaddorna till flockdjur som följer en ledar-sköldpadda. Skapa en ny sköldpaddsklass DependentTurtle som skickar changed-meddelanden då den flyttar sig. Lägg också till en update-metod som vrider sköldpaddan mot den sköldpadda som flyttade sig. Nu kommer alla sköldpaddor röra sig mot sin ledare. Prova!

10 Strömmar

I det här kapitlet besvaras bland annat följande frågor:

- Vad är en ström?
- Hur konstrueras slumptal?
- Hur styr ett objekt sin presentation vid **print it**?
- Hur kan objekt lagras på sekundärminne och senare återskapas?

I Smalltalk finns det en stor mängd klasser som, vid sidan om Collection, hanterar objekt av variabel typ. En viktig familj av detta slag är klassen Stream med subklasser. Dessa kan användas för att på ett smidigt sätt positionera, läsa och skriva på objekt av indexerbar typ, tex på instanser av Collection.

Strömmar är särskilt användbara vid syntaxanalys (eng parsing) och är direkt nödvändiga vid filbehandling och annan typ av extern kommunikation, tex via portar och terminalemulatorer.

10.1 Inledning

Strömmar är väsentligen av tre olika typer nämligen:

- *läs* – med hjälp av strömmar av lästyp kan positionering och läsning av information från indexerbara objekt ske. Det går däremot inte att lägga till nya objekt.
- *skriv* – med denna typ av ström går det endast att lägga till ny information i slutet.
- *direktåtkomst* dvs läs-och-skriv, ibland också kallad indexerbar (eng randomized) – denna ström kombinerar läs- och skrivströmmens möjligheter. Det går att både positionera, läsa och skriva över hela det indexerbara objekt som strömmen verkar på. Det finns också varianter, som tex strömmar som är positionerbara för läsning men med vilka all skrivning alltid sker i slutet av det underliggande objektet. Dessa strömmar brukar kallas för läs-och-lägg-till-strömmar (eng read-append).

Vad kan vi öppna en ström på?

Vi kan öppna en ström på en behållare som har en ordning definierad mellan sina element, dvs en subklass till SequenceableCollection som tex Array, OrderedCollection eller String.

Även klassen Random som används för att generera slumptal tillhör familjen strömmar. Detta trots att klassen logiskt tillhör de numeriska klasserna.

Hur allokeras utrymme i behållaren?

Om vi tex öppnar en ström på ett behållare som allokeras med en fast storlek, tex en Array, och sedan vill skriva nya objekt på den, hur gör vi då? Svaret är att den aktuella strömmen måste se till att behållaren växer vid behov. Detta gör att vi utan att bekymra oss även kan behandla statiska behållare som objekt med dynamisk tillväxt om vi applicerar strömmar på dem.

Hierarki

Klasshierarkin för klassen Stream med subklasser ser ut som följer:

```
Object
  Stream
    ExternalDatabaseAnswerStream
    PeekableStream
      PositionableStream
        ExternalStream
          ..."I avsnittet filhantering"
        InternalStream
          ..."Mer detaljer nedan"
    Random
```

Figur 10.1 Stream med subklasser

Som synes finns det förutom klassen Random två undergrenar med subklasser till ExternalStream och InternalStream. I detta kapitel ska vi främst behandla InternalStream med subklasser. De externa strömmarna för hantering av filer och portar är till stora delar lika med de interna. Detta medför bla att en beskrivning av interna strömmar också ger grunderna för hantering av externa strömmar. Klassen ExternalStream med subklasser behandlas i kapitel 11.

Inledande exempel

Innan vi ger oss in på detaljer för de olika ström-typerna ger vi några korta typexempel. Detaljerna förklaras senare i de efterföljande underavsnitten.

Först skapar vi en enkel ström för läsning applicerad på en vektor.

s := #(a b c d e) readStream.		skapa en läsström på en vektor
s next.	⟹*#a*	läs nästa tecken
s peek.	⟹*#b*	vad pekar strömpekaren på?
s skip: 1.		flytta pekaren ett steg
s next	⟹*#c*	läs nästa element

Figur 10.2 Exempel: läsström

Sedan konstruerar vi en ström för både läsning och skrivning på en från början tom sträng:

s := ReadWriteStream on: ''.		skapa läs- och skrivbar ström på en tom sträng
s nextPut: $a.		skriv tecknet a på strömmen
s nextPutAll: 'bcde'.		skriv en hel sträng på strömmen
s reset.		sätt läspekaren till början
s next: 2.	⟹*'ab'*	läs de två följande elementen
s upToEnd	⟹*'cde'*	läs resten

Figur 10.3 Exempel: läs- och skrivström

10.2 Interna strömmar

En intern ström läser eller skriver på ett indexerbart objekt som befinner sig i primärminnet. Klasshierarkin för dessa strömmar ser ut som följer:

```
InternalStream
  ReadStream
  WriteStream
    ReadWriteStream
      ByteCodeReadWriteStream
    TextStream
```

Figur 10.4 Interna strömmar

Där klassen InternalStream främst specialiserar visst beteende från klassen PositionableStream. Detta för att möta de olika interna strömmarnas behov av att tex öka i storlek om de blir fulla. Klasserna ReadStream, WriteStream och ReadWriteStream gör vad som namnen antyder, nämligen skapar en enbart läsbar, en enbart skrivbar eller alternativt en både läs- och skrivbar ström. Rent tekniskt implementeras dessa möjligheter genom att lämpliga metoder läggs till eller omdefinieras i subklasser. I vissa fall använder subklasserna shouldNotImplement för att förhindra att icke tillämpbara metoder används. Klassen TextStream hanterar strömmar över textsträngar som har viss typografisk information, såsom fetstil, färg och understrykning, knuten till sig.

Konstruktion

En ström konstrueras antingen genom ett meddelande till någon av de olika strömklasserna eller genom att direkt be ett objekt av de ordnade behållarklaserna (Array, String, OrderedCollection mfl) om att skapa en ström.

Grunder

Det finns två grundläggande konstruktörer för att skapa strömmar.

on:	ger en ny ström som ”strömmar” över argumentet. Positionspekaren sätts till noll, läspekaren sätts till början och skrivpekaren till slutet.
with:	ger en ny ström som ”strömmar” över argumentet. Positions-, läs- och skrivpekare sätts till slutet.

Figur 10.5 Konstruktörer för strömmar

Skillnaden mellan dessa två metoder kan tyckas hårfin. För en lässström så innebär konstruktionen med on: att positionspekaren sätts till början. Med with: däremot sätts positionspekaren till slutet.

```
r := ReadStream on: #(#jessica #hanna #klara #jesper #hugo #jonatan).
r next
```
⟹*#jessica*

```
r := ReadStream with: #(#jessica #hanna #klara #jesper #hugo #jonatan).
r next
```
⟹*nil*

För en skrivström är det ofta avgörande att välja rätt konstruktör, vilket framgår av följande exempel:

```
w := WriteStream on: #(#jessica #hanna #klara #jesper #hugo #jonatan).
w nextPut: #kalle.
w contents
```
⟹*#(#kalle)*

```
w := WriteStream with: #(#jessica #hanna #klara #jesper #hugo #jonatan).
w nextPut: #kalle.
w contents
```
⟹*#(#jessica #hanna #klara #jesper #hugo #jonatan #kalle)*

Här ser vi att med on: så betraktas strömmen som tom men med with: så fortsätter vi att skriva i slutet.

Konstruktion av strömmar på behållare

För att konstruera en läs- eller skrivström finns det två unära meddelanden som förstås av de ordnade behållarklasserna, dvs SequenceableCollection med subklasser.

readStream	skapa en lässström på mottagande objekt
writeStream	skapa en skrivström på mottagande objekt

Figur 10.6 Konstruera en ström givet en behållare

Vill man däremot skapa en ström för både läsning och skrivning så måste klassen ReadWriteStream användas direkt, med någon av dess konstruktionsmetoder och önskat behållarobjekt som argument.

Både readStream och writeStream är implementerade med konstruktören on:, så positionen blir satt till början i båda fallen.

Delmängder

Det är också möjligt att konstruera en ström på en delmängd av en behållare.

on:from:to:	öppna en ström som arbetar på en kopia av ett utsnitt av en behållare
with:from:to:	öppna en ström på en delmängd (av en icke föränderlig) behållare

Figur 10.7 Ström på delmängd av behållare

Om vi tex vill skapa en lässström på element två tom fyra i en vektor kan det hela se ut så här:

```
stream := ReadStream on: #(1 2 3 4 5) from: 2 to: 4
```

Hantering

Efter det att en ström har skapats så kan den läsas, skrivas, positioneras och sökningar på en mängd olika sätt kan utföras. Vad som är möjligt att göra med strömmen beror till stor del av dess typ. Det går tex inte att skriva på en ström skapad enbart för läsning eller läsa enstaka element från en ström skapad för enbart skrivning. Hela innehållet kan dock alltid läsas mha meddelandet contents.

Vi har nedan gjort en kort beskrivning av användbara operationer på strömmar, där de mest grundläggande kommer först och de lite mer speciella senare.

Grundläggande metoder på strömmar

De mest grundläggande operationerna på strömmar, som de flesta av de andra mer avancerade metoderna sedan i sin tur använder, är följande:

next	nästa element
nextPut: anObject	skriv ett enstaka element
nextPutAll: aCollection	skriv mängd med element
next: anInteger	läs det antal element som argumentet anger
position	strömpekarens position som ett heltal
position: anInteger	flytta strömpekaren till den position som ges av argumentet
size	antalet element på strömmen

Figur 10.8 Grundläggande operationer på strömmar

skip: anInteger	hoppa förbi det antal element som argumentet anger
peek	läs elementet vid pekaren, utan att flytta pekarens position
atEnd	är strömpekaren vid slutet?
do: aBlock	utför ett block på alla element från aktuell position och fram till slutet
contents	hela innehållet
reset	flytta strömpekaren till början
setToEnd	flytta strömpekaren till slutet
upToEnd	alla element från aktuell position tom slutet

Figur 10.8 Grundläggande operationer på strömmar

Vi illustrerar det hela med en lässström på en vektor innehållande ett antal namn, givna som symboler.

```
namn := #(björn per olle nina åsa katarina jessica jesper jonatan
          hanna klara hugo).
ström := namn readStream.                                  skapa lässström
ström position.          ⟹0                                aktuell position
ström size.              ⟹12                               storlek
ström next.              ⟹#björn                           nästa element
ström next: 2.           ⟹#(#per #olle)                    de två följande elementen
ström position.          ⟹3                                positionen är nu
ström peek.              ⟹#nina                            vad pekas ut
ström position.          ⟹3                                nu ”är vi” fortfarande i
                                                           position 3
ström skip: 3.                                             flytta tre element framåt
ström peek.              ⟹#jessica                         vad pekas ut nu?
ström atEnd.             ⟹false                            är vi i slutet?
ström upToEnd.           ⟹#(#jessica #jesper #jonatan      läs från aktuell position
                         #hanna #klara #hugo)              fram tom slutet
```

Figur 10.9 Exempel: operationer på lässström

Meddelandet do: är nästan ekvivalent med de olika behållarnas motsvarande operation. Skillnaden är att do: för behållare utför blocket på alla element medan do: på strömmar startar i aktuell position. Bägge fortsätter till slutet av behållaren.

Exempel: Läs- respektive skrivström

Vi ger ett exempel där vi först konstruerar en ström på ett intervall från 5 till 50 med ett steg 5. Vi flyttar sedan förbi de fem första elementen, dvs fram till 30. Slutligen bildar vi en skrivström och skriver alla kvadrater av elementen from aktuell position tom slutet av läsströmmen.

```
numbers := 5 to: 50 by: 5.
numberStream := numbers readStream.
numberStream skip: 5.
squareStream := #() writeStream.
numberStream do: [:x | squareStream nextPut: x squared].
squareStream contents.
```

⟹*#(900 1225 1600 2025 2500)*

Exempel: Skrivström

Vi konstruerar ytterligare ett enkelt exempel med en skrivström som använder do: och contents.

```
table := Dictionary new.
table at: 0 put: 'noll'; at: 1 put: 'ett'; at: 2 put: 'två'; at: 3 put: 3.
stream := #() writeStream.
-1 to: 4 do: [:i | stream nextPut: (table at: i ifAbsent: [i])].
stream contents
```

⟹*#(-1 'noll' 'ett' 'två' 3 4)*

Där vi har definierat en katalog med ett antal nyckel-värdepar. Sedan skapar vi ett intervall med start = -1 och slut = 4 och använder katalogen för att slå upp respektive värde och skriva resultatet på skrivströmmen. Som framgår används nyckeln om inte motsvarande värde hittas i katalogen.

Sökmetoder på strömmar

Det finns också ett flertal sätt att söka efter eller läsa flera objekt samtidigt från strömmar.

through: anObject	svara med en delmängd från aktuell position tom nästa förekomst av anObject. Om inget sådant objekt hittas svara med resten av strömmen.
upTo: anObject	som through: fast utan att anObject ingår i resultatet. Strömpekaren blir placerad efter anObject.

Figur 10.10 Metoder för att söka objekt

skipThrough: anObject	flytta pekaren till anObject, till slutet om anObject inte finns.
skipUpTo: anObject	sök efter anObject, pekaren flyttas till slutet om anObject inte finns.
peekFor: anObject	är nästa element på strömmen lika med anObject?
nextMatchFor: anObject	läs nästa element och svara om det är lika med anObject
throughAll: aCollection	läs till och med nästa sekvens av objekten i aCollection. Om ingen sådan sekvens existerar returneras resten av strömmen.
upToAll: aCollection	som throughAll: fast icke inklusive
skipToAll: aCollection	hoppa framåt till nästa förekomst av aCollection. Om träff placera positionspekaren före aCollection, annars i slutet av strömmen.
nextAvailable: anInteger	nästa anInteger antal element. Om det inte finns angivet antal läses så många som möjligt.

Figur 10.10 Metoder för att söka objekt

Vi fortsätter att arbeta med namnströmmen från figur 10.9, för att illustrera några av dessa operationer också.

ström reset.		sätt pekaren till början
ström skipUpTo: #nina.		flytta till nästa förekomst
ström through: #katarina.	⟹*#(#nina #åsa #katarina)*	läs till och med
ström peek.	⟹*#jessica*	vad pekas ut?
ström position: ström position + 3.		ändra position med absolut angivelse
ström skip: -2.		ändra position, med relativ angivelse (den här gången flyttar vi mot början)
ström nextAvailable: 50.	⟹*#(#jesper #jonatan #hanna #klara #hugo)*	nästa 50 element eller så många som möjligt

Figur 10.11 Exempel: Flytta och söka på ström

Det går också att enkelt skriva ett visst objekt ett upprepat antal gånger med ett enda meddelande (next:put:).

next: anInteger put: anObject	skriv anObject det antal gånger som anges av anInteger

Figur 10.12 Skriv ett objekt ett upprepat antal gånger på strömmen

Här är ett exempel där vi både skriver ett element upprepade antal gånger och konstruerar en ström på en delmängd av givna indata.

```
in := #(1 2 3 4 5 6 7) readStream.
filter1 := #() writeStream.
[in nextMatchFor: 3] whileFalse.                    Vi söker efter nästa 3'a
in do: [:x | filter1 next: x put: x squared].       Skriv 4 i kvadrat fyra gånger, 5
                                                    i kvadrat fem gånger, osv.
filter1 contents.                                   ⟹#(16 16 16 16 25 25 25 25 25 36
                                                    36 36 36 36 36 49 49 49 49 49 49
                                                    49)
filter2 := ReadWriteStream                          Vi arbetar på ett utsnitt. I det
        with: filter1 contents from: 5 to: 15.      här fallet from första 25:an
                                                    tom sista 36:an
filter2 reset.
[filter2 atEnd]
    whileFalse: [| nxt |
        nxt := filter2 next.                        Läs nästa tal
        [nxt = filter2 peek]                        så länge som det är samma
            whileTrue:                              som nästa tal så skriver vi över
                [filter2 skip: -1.                  med aktuellt värde plus ett.
                filter2 nextPut: filter2 next + 1]].
filter2 contents                                    ⟹#(25 26 27 28 29 36 37 38 39 40
                                                    41)
```

Figur 10.13 Exempel: Filter

Skiljetecken

Det finns också metoder som gör det enkelt att hantera skiljetecken.

cr	skriv en vagnretur
space	skriv ett blanktecken
tab	skriv ett tabuleringstecken
tab: anInteger	skriv anInteger antal tabuleringstecken
crtab	skriv vagnretur följt av tabuleringstecken
skipSeparators	flytta strömpekaren förbi alla eventuellt direkt efterföljande skiljetecken

Figur 10.14 Skiljetecken

Ett exempel på dessa operationer är följande, där vi antar att vi har en liten databas med nyckel- värdepar. Det har smugit sig in några extra blank-, tabulerings- och nyradstecken i data som vi filtrerar bort. På resultatströmmen skriver vi blanka, tabuleringar och nya rader med hjälp av metoder från figur 10.14.

```
data := ' nyckelEtt  värdeEtt   nyckelTvå

värdeTvå  nyckelTre   värdeTre
'.
in := data readStream.
out := String new writeStream.
[in skipSeparators; atEnd]                    Skippa skiljetecken; är vi i slutet?
    whileFalse: [| nxt |
        [(nxt := in next) isSeparator not] whileTrue: [out nextPut: nxt].
        in skipSeparators.          Ta bort skiljetecken mellan nyckel och värde
        out space; nextPutAll: '->'; tab.
        [in atEnd not
            and: [(nxt := in next) isSeparator not]] whileTrue: [out nextPut: nxt].
        in atEnd not ifTrue: [out cr]].              Radbyte, fast ej efter sista värdet
out contents
```

⇒

'nyckelEtt ->värdeEtt
nyckelTvå ->värdeTvå
nyckelTre ->värdeTre'

Exempel: Utbyte av ord

Det här exemplet visar hur vi med hjälp av ett Dictionary och strömmar enkelt kan byta ut förekomster av vissa ord mot andra i en sträng.

```
"Vi skapar en katalog"
items := Dictionary new.
```

```
"Vi definierar några nycklar med värden"
items at: '1' put: 'ett'; at: '2' put: 'två'; at: '3'  put: 'tre'; at: '5'  put: 'fem';at: '9' put:
'nio'; at: '+' put: 'plus';  at: '-' put: 'minus'; at: '*' put: 'gånger'; at: '=' put: 'lika med'.

"Vi konstruerar en sträng med lite text där vi senare ska byta ut 'kända'
siffervärden mot motsvarande texter"
string := 'Så följande 5 - 2 * 3 = 9 och alltså inte = -1 kan kanske förklara 1 och
annat
'.
inStream := string readStream.
outStream := String new writeStream.
[inStream atEnd]                                          "Har vi nått slutet?"
    whileFalse: [|  item |
        "Vi använder item för att mellanlagra nästa ord som hittas på
        inStream"
        item := String new writeStream.
        "Så länge som nästa tecken i inStream är ett skiljetecken så skriver vi
        det direkt på outStream"
        [inStream atEnd not and: [inStream peek isSeparator]]
            whileTrue: [outStream nextPut: inStream next].
        "Så länge som det som följer på inStream är alfabetiska tecken så
        bygger vi upp nästa ord, dvs item"
        [inStream atEnd not and: [inStream peek isSeparator not]]
            whileTrue: [item nextPut: inStream next].
        "Om vi inte nådde slutet så finns ett ord i item"
        item size > 0
            ifTrue: ["Slå upp ordet i katalogen. Finns det inte så
                    använd item direkt"
                    outStream nextPutAll: (items at: item contents
                                                ifAbsent: [item contents])]].
outStream contents
```

⟹*'Så följande fem minus två gånger tre lika med nio och alltså inte lika med -1 kan kanske förklara ett och annat*

Mindre vanliga metoder på strömmar

Ibland kan man behöva kontrollera en ströms typ eller skriva innehållet av den direkt på en annan behållare med start i en viss position. Det finns några metoder som understödjer detta.

isReadable	är strömmen läsbar, dvs implementeras next:
isWriteable	är strömmen skrivbar, dvs implementeras nextPut:
next:into:startingAt:	skriv det antal element som anges som argument till next: på en behållare som är argument till into:. Starta i den behållarposition som anges av heltalsargumentet till startingAt:. Om indexgränserna överskrids genereras ett felavbrott.
next:putAll:startingAt:	läs det antal element som anges som argument till next: från behållaren efter nyckelordet putAll:. Starta i den behållarposition som anges av argumentet till startingAt:.
nextAvailable:into:startingAt:	som next:into:startingAt: fast ger ej felavbrott om indexgränserna överskrids

Figur 10.15 Fler operationer på strömmar

Stänga och tömma strömmar

Det finns ett par metoder som är mer inriktade på externa strömmar än de tidigare beskrivna metoderna. Dessa metoder gör ingenting på en intern ström, men om en extern ström används så får de effekt. Men för att göra det enkelt att ersätta en intern ström med en extern dito finns dem ändå definierade för båda typerna.

flush	töm innehållet av strömbufferten (se tex till att innehållet verkligen skrivs ut på filen)
close	stäng strömmen (vanligen filen)

Figur 10.16 Tömma buffert eller stänga ström

10.3 Objektens presentation på skärmen

Alla objekt i Smalltalk använder strömmar på ett eller annat sätt. Oftast behöver man som programmerare inte bekymra sig om när eller hur, men vissa grundläggande mekanismer bör man känna till. Det kan bla vara bra att veta hur klasserna definierar hur dess instanser presenteras i text- eller listfönster.

Klassen definierar hur dess objekt presenteras

Varje klass definierar hur dess instanser ska presentera sig som text på skärmen. Om klassen inte definierar någon egen metod för detta så används den aktuella superklassens metod. Om inget annat anges kommer Object's metod för presentation att användas. Detta medför också att alla klasser har något sätt att presentera sina objekt. Metoden i klassen Object definerar presentationen som artikel (a, an) följt av klassnamn. Tex så presenterar sig en instans av Object som *an Object*, en instans av ReadStream som *a ReadStream*, men strängen 'test' som *test* och vektorn #(1 2 3) helt enkelt som *#(1 2 3)*. Orsaken är att presentationsmetoden för dessa klasser är omdefinierad.

Här följer en mer detaljerad beskrivning av hur klasser definierar hur dess instanser presenteras på skärmen.

printOn: aStream	Definierar hur ett objekt ska presentera sig på en ström.
printString	Definierar hur ett objekt ska skrivas som text. Detta meddelande används då ett objekt skrivs i ett textfönster. Metoden använder i sin tur metoden printOn: för att skriva ut de enskilda objekten
displayString	Definierar hur ett objekt ska presenteras med (eventuellt) lite mer textuell information, tex fetstil. Exempelvis så använder listvyer detta meddelande för att presentera sina objekt. I klassen Object är displayString implementerad genom att anropa printString.

Figur 10.17 Metoder för utskrift av objekt

För att uppmana ett objekt att presentera sig själv på den aktuella strömmen finns meddelandet print:, definierad i Stream. Detta meddelande ser man oftast i samband med att objekt skrivs på strömmen med hjälp av kaskadmeddelanden.

print: anObject	uppmana anObject att använda printOn: för att skriva sig själv på strömmen

Figur 10.18 Metod för att uppmana objekt att skriva sig på ström

Exempel: Bilregister

Vi konstruerar en enkel klass tänkt att representera bilar i ett register. Instanserna lagrar information om en bils registreringsnummer, tillverkningsår och modell.

```
Object subclass: #Car
    instanceVariableNames: 'registrationNumber year model '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Print-and-store'
```

Åtkomstmetoderna för de olika instansvariablerna ser ut precis som vanligt så vi beskriver inte dem.

KLASSMETODER

instance creation

```
registrationNumber: aRegistrationNumber year: aYear model: aModel
    ^self new
        registrationNumber: aRegistrationNumber
        year: aYear
        model: aModel
```

INSTANSMETODER

printing

```
registrationNumber: aRegistrationNumber year: aYear model: aModel
    self registrationNumber: aRegistrationNumber.
    self year: aYear.
    self model: aModel

printOn: aStream
    "Först skriver vi 30 minustecken"
    aStream next: 30 put: $-.
    aStream cr.
    "Sen skriver vi en ledtext och skriver registreringsnumret på aStream"
    aStream nextPutAll: 'Registreringsnummer:'; space.
    self registrationNumber printOn: aStream.

    "Vi kan använda print: istället för printOn:"
    aStream crtab: 2; nextPutAll: 'modell:'; space; print: self model.
    aStream crtab: 2; nextPutAll: 'år:'; space; print: self year
```

```
displayString
    | output |
    output := '' writeStream.
    output nextPutAll: 'reg: '; nextPutAll: self registrationNumber; space;
    nextPutAll: 'modell: '; nextPutAll: self model; space; nextPutAll: 'år: '; print: self
    year.
    ^output contents
```

Vi prövar det hela genom att skriva ett litet test med utmatning i utmatningsfönstret.

```
car1 := Car registrationNumber: 'OOPFUN' year: 1995 model: 'VolWork'.
car2 := Car registrationNumber: 'BASFUN' year: 1995 model: 'Object Engine'.
Transcript cr; print: car1; cr; print: car2; endEntry
```

⇒

```
------------------------------
Registreringsnummer: 'OOPFUN'
        modell: 'VolWork'
        år: 1995
------------------------------
Registreringsnummer: 'BASFUN'
        modell: 'Object Engine'
        år: 1995
```

Resultatet av displayString illustrerar vi genom att skicka motsvarande meddelande till car1:

```
car1 displayString
```

⇒*'reg: OOPFUN modell: VolWork år: 1995'*

10.4 Lagra och återskapa objekt

Nu ska vi diskutera hur klasserna definierar dess objekts externa representation samt hur instanser sedan återskapas från dessa. Mer avancerade sätt att åstadkomma detta beskrivs i avsnitt 11.4 om BOSS.

Lagra och återskapa

Varje objekts klass har möjlighet att på ett smidigt sätt definiera hur dess instanser ska sparas på och återskapas från strömmar. Normalt finns det ingen anledning att definiera om dessa metoder, men för tex cirkulära listor är detta nödvändigt. Orsaken är att Object's metoder inte är förberedda för sådana objekt, dvs de löser inte upp cirkuläriteter.

Ett objekt använder en ström för extern lagring och återskapande

Ekvivalent med att en klass beskriver hur dess objekt ska presentera sig så beskriver den också hur dess instanser ska lagras på extern form, tex för att senare återskapas från sekundärminne. Vidare ansvarar också ett objekts klass för hur instanser sparade på en ström ska återskapas, dvs instansieras, igen.

Lagring på eller återskapande från ström

Det är väsentligen två olika metoder inblandade vid ett objekts lagring respektive återskapande från en ström. I många fall duger klassen Object's metoder, men om en effektivare eller mer lättläst form önskas kan metoderna definieras om i respektive klass.

storeOn: aStream	Instansmetod som definierar hur ett objekt ska lagras på en ström
readFrom: aStream	Klassmetod som definierar hur objekt ur den aktuella klassen ska återskapas efter att de har lagrats med storeOn:.

Figur 10.19 Lagra och återskapa objekt

Analogt med att meddelandet print: uppmanar ett objekt att presentera sig själv på den aktuella strömmen så uppmanar store: objektet att använda storeOn: för att lagra sig själv.

store: anObject	uppmana anObject att använda storeOn: för att lagra sig själv på strömmen

Figur 10.20 Uppmana objekt att lagra sig på ström

Exempel: Lagra bilar med metoder definierade i Object

Genom att använda de metoder som är ärvda från Object kan vi lagra respektive återskapa instanser av Car.

```
stream := '' writeStream.
car1 storeOn: stream.
stream contents
```

⟹*'(Car basicNew instVarAt: 1 put: ''OOPFUN''; instVarAt: 2 put: 1995; instVarAt: 3 put: ''VolWork''; yourself)'*

```
inStream := stream contents readStream.
carX := Car readFrom: inStream.
carX displayString
```

⟹*'reg: OOPFUN modell: VolWork år: 1995'*

Om vi vill att bilarna ska lagras på ett mer kompakt format, som också automatiskt ger väldefinerade gränser mellan olika bilar lagrade på gemensamma strömmar, så måste vi definiera om metoderna storeOn: och readFrom: i Car.

Exempel: Lagra bilar med egna metoder

Vi skriver om storeOn:. För att illustrera de olika möjligheterna lagras instansvariabler direkt med storeOn: men också indirekt med store:. Vi använder utropstecken (!) som avskiljare mellan olika bilar på strömmen.

```
storeOn: aStream
    "Lagrar self på aStream"
    self registrationNumber storeOn: aStream.
    aStream space.
    aStream store: self year; space.
    aStream store: self model; space.
    aStream nextPut: $!
```

Nu kan vi enkelt på ett kompakt format lagra flera olika bilar på samma ström.

```
car1 storeOn: stream.
car2 storeOn: stream.
garage := stream contents
```

⟹*'"OOPFUN" 1995 "VolWork" !"BASFUN" 1995 "Object Engine" !'*

Exempel: Läsa in bilar

Nu skriver vi klassmetoden readFrom: i Car class. Observera speciellt hur avskiljaren, dvs utropstecknet, används för att definiera hur långt vi ska läsa på strömmen.

```
readFrom: aStream
    "Återskapar ett objekt från aStream"
    | reg year model |
    reg := aStream upTo: Character space.
    year := aStream upTo: Character space.
    model := aStream upTo: $!.
    ^self
        registrationNumber: reg
        year: year
        model: model
```

Nu skapar vi en lässström på variabeln garage från det föregående exemplet och lagrar bilarna i en OrderedCollection.

```
in := garage readStream.
cars := OrderedCollection new.
[in atEnd] whileFalse: [cars add: (Car readFrom: in)].
```



```
    cars
⟹OrderedCollection (------------------------------
Registreringsnummer: '"OOPFUN"'
    modell: '"VolWork" '
    år: '1995' ------------------------------
Registreringsnummer: '"BASFUN"'
    modell: '"Object Engine" '
    år: '1995')
```

Lagring på eller återskapande från sträng

För att pröva de två metoderna för att lagra respektive återskapa objekt kan vi använda strängar som lagringsmedium. Då används följande två metoder, den första för att lagra och den andra för att återskapa objekten.

storeString	Den här instansmetoden applicerar metoden storeOn: på det mottagande objektet och returnerar dess innehåll i form av en sträng
readFromString: aString	Den här klassmetoden återskapar ett objekt från aString. En ström skapas på strängen och metoden readFrom: appliceras på den.

Figur 10.21 Lagra eller läsa objekt från strängar

Behovet av dessa metoder finns främst för objekt som hanteras internt i minnet. På filer blir resultatet strängar direkt, via storeOn: och readFrom:, och resultatet kan därigenom direkt undersökas eller modifieras med en konventionell filhanterare.

Spara på fil

Det är lika enkelt att lagra objekt på en extern ström, vanligen kopplad till en fil, som en intern ström. Ur objektets synvinkel märks ingen skillnad. Det enda som egentligen skiljer är att vid ”ihopkopplingen” ges en extern istället för en intern ström. På motsvarande sätt kan objekt lagrade på extern form läsas på precis samma sätt som om de hade varit lagrade internt. För en utförligare beskrivning se kapitel 11.

10.5 Slumptal med klassen Random

En instans av klassen Random tillhandahåller en oändlig ström av likformigt fördelade slumptal i det halvöppna intervallet [0, 1); dvs 0 <= r < 1, där r är nästa slumptal på strömmen.

Instanser av klassen kan skapas genom att skicka meddelandet fromGenerator:seededWith: med ett generatornummer mellan 1 och 7 samt ett av användaren givet numeriskt frö, som i:

```
randomNumber := Random fromGenerator: 5 seededWith: 12342
```

Det går också att använda meddelandet new för att skapa en ny slumptalsgenerator. I så fall ger systemet oss den av generatorerna som är definierad som skönsvärde och bildar fröet från systemklockan. Vi illustrerar detta användningssätt med följande lilla exempel där vi konstruerar tio slumptal mellan 10 och 100 som i tur och ordning placeras i en behållare.

```
| randomNumber collectionOfNumbers lowerBound upperBound |
randomNumber := Random new.
collectionOfNumbers := OrderedCollection new.
lowerBound := 10.
upperBound := 100.
10 timesRepeat: [
    collectionOfNumbers
            add: (randomNumber next *
                    (upperBound - lowerBound) +
                        lowerBound) truncated].
collectionOfNumbers
```

Svaret kan tex bli:

⟹*OrderedCollection (41 59 92 24 83 36 88 69 67 67)*

Sett ur klassen Random's synvinkel är det intressanta att slumptalsgeneratorn konstruerades med klassmeddelandet new. Därmed gavs en mindre förutsägbar sekvens av slumptal än om något speciellt frö och generator hade angetts. Som också framgår används meddelandet next för att generera nya slumptal. I vissa sammanhang är det dock viktigt med slumptalssekvenser som är *kontrollerbara* där både frö och generator ges ”manuellt”. Ett visst frö i kombination med en viss generator ger nämligen alltid samma slumptalsserie.

10.6 Exempel

Från romersk till decimal representation

Vi skriver en klass för att konvertera tal givna med romerska siffror till tal med decimal representation. Vi använder en klassvariabel, RomanSymbolsToDecimal, för att beskriva vilket decimalt värde en viss romersk siffra har. För att också se till att klassen initieras om vi gör **file in** på den skriver vi en klassmetod initialize som anropar metoden initializeTable som i sin tur initierar klassvariabeln. Då speciella konstruktörer ska användas tillåter vi inte att klassen konstrueras med meddelandet new.

Motorn i klassen är instansmetoden convertToDecimal:, i vilken vi använder en ström och gör en del icke fullständiga kontroller av om de romerska talet är skrivet på ett riktigt sätt. I övning 10.1 gör vi en mer fullständig kontroll av om formatet är riktigt.

```
Object subclass: #RomanToDecimalConverter
    instanceVariableNames: ''
    classVariableNames: 'RomanSymbolsToDecimal '
    poolDictionaries: ''
    category: 'bok-1-numeric'

KLASSMETODER

class initialization

initializeTable
    "Skapar en katalog med de romerska siffrorna"
    RomanSymbolsToDecimal := Dictionary new.
    RomanSymbolsToDecimal at: $I put: 1; at: $V put: 5; at: $X put: 10; at: $L put:
    50; at: $C put: 100; at: $D put: 500; at: $M put: 1000

initialize
    self initializeTable

instance creation

new
    self shouldNotImplement

convert: aRomanString
    ^self basicNew convertToDecimal: aRomanString

INSTANSMETODER

accessing

convertToDecimal: aRomanString
    "Konverterar aRomanString till ett decimalt tal"
    | stream decimalValue |
    stream := aRomanString readStream.
    decimalValue := 0.
    [stream atEnd] whileFalse:
```

```
        [| digitValue |
        digitValue := self romanCharacterToDecimal: stream next.
        (stream atEnd not
                and: [digitValue < (self romanCharacterToDecimal: stream peek)])
            ifTrue: [digitValue := digitValue negated].
        decimalValue := decimalValue + digitValue].
    ^decimalValue
```

private

romanCharacterToDecimal: aRomanCharacter

```
    "Slå upp tecknet i tabellen och returnera motsvarande decimala värde"
    ^RomanSymbolsToDecimal at: aRomanCharacter asUppercase
```

Vi prövar klassen.

```
RomanToDecimalConverter convert: 'xxiv'.
```

⟹*24*

```
RomanToDecimalConverter convert: 'MCMLXXXXV'.
```

⟹*1995*

Enkel syntaxanalysator, för epost-liknande texter

Nu ska vi konstruera en enkel syntaxanalysator för texter med märkord, av typ elektronisk post (epost). Vi gör det hela i ett arbetsfönster men kan enkelt vidarutveckla det hela till en mer fullständig klass när vi har sett att det hela fungerar som vi vill.

I många system för elektroniks post brukar ett brev eller inlägg inledas med ett litet texthuvud bestående av ett antal rubrikord följt av beskrivande texter. Rubrikorden avlutas med en speciell symbol, ofta ett kolon.

I koden som följer gör vi:

1 initiering
- vi skapar en testtext som vi tilldelar till variabeln text
- därefter skapar vi en lässström, stream, på text
- vi initierar också dict (instans av Dictionary) avsedd att hålla i resultatet av analysen i form av nyckel-värde-par.

2 itererar
så länge som vi inte har nått slutet på strömmen (stream atEnd) läser vi först nästa rubrik samt efterföljande text och lägger den i dict

```
| text  stream dict delimiter |
text := 'ämne: Illustrera Stream från: Oss Alla till: Läsarna datum: 9 October 2002
data: Genom att använda en ström och en instans dict, ett Dictionary, ska vi
stycka upp meddelandets innehåll. För varje märke konstruerar vi en nyckel i dict
och använder texten upp till nästa märke som värde. Vi tar också bort en del
onödiga skiljetecken.'.
```

```
stream := text readStream.
dict := Dictionary new.
delimiter := $:.                          delimiter definierar vad som
[stream atEnd]                            avslutar ett nycklord
    whileFalse:
        [| tag dataPosStart |
        stream skipSeparators.            Ta bort skiljetecken
        tag := stream upTo: delimiter.    Läs nästa nyckelord
        stream skipSeparators.
        dataPosStart := stream position.  Vi kommer ihåg var data börjar
        "Vi söker efter fler nyckelord"
        [stream next ~= delimiter and: [stream atEnd not]] whileTrue.
        stream atEnd
            ifTrue:
                ["Vi hittade inga fler märken, vilket medför att resten är data"
                stream position: dataPosStart.
                dict at: tag put: stream upToEnd]
            ifFalse:
                [ | newPos |
                [stream skip: -1.
                stream peek isSeparator] whileFalse.
                [stream peek isSeparator] whileTrue: [stream skip: -1].
                newPos := stream position.
                stream position: dataPosStart.
                dict at: tag
                    put: (stream next: newPos - dataPosStart + 1)]]
```

Resultatet kommer att se ut enligt följande:

⟹*Dictionary ('ämne'->'Illustrera Stream' 'till'->'Läsarna' 'data'->'Genom att använda en ström och en instans dict, ett Dictionary, ska vi stycka upp meddelandets innehåll. För varje märke konstruerar vi en nyckel i dict och använder texten upp till nästa märke som värde. Vi tar också bort en del onödiga skiljetecken.' 'från'->'Oss Alla' 'datum'->'9 oktober 2002')*

Textreferenser

Det är vanligt att texter förses med ett index. Vi ska nu konstrura ett enkelt ”system” där vi kan märka en text med indexord eller referenser. Vi skriver en analysator för texten. Denna analys resulterar i två olika resultat.

1 Text utan indexord, outStream contents
  Detta är den egentliga, icke indexmärkta, texten, dvs då alla indexord är bortfiltrerade.

2 Behållare med alla indexords placering i, tags
  När ett indexord påträffas läggs detta i en speciell lista med ett numeriskt värde som anger var i den egentliga texten ordet finns.

För att indexorden ska vara lätta att identifiera placeras de mellan ett par speciella märktecken. För att detta tecken enkelt ska kunna bytas ut använder vi oss av en lokal variabel, tag. För att det ska bli så enkelt som möjligt att märka texten väljer vi att använda @ som märktecken. I en slutlig version kan detta tecken mycket väl ersättas med ett mer speciellt tecken (som ett styrtecken eller liknande).

Testversion i ett arbetsfönster

Vi gör det hela i ett arbetsfönster och låter den intresserade läsaren konstruera klasser för att göra motsvarande arbete. Även om vi använder ett arbetsfönster och vi vet att den kod vi skriver inte är den slutgiltiga så tjänar man på att vara strukturerad och isolera funktionalitet. Både för att få en mer överskådlig lösning och för att lära känna problemets struktur för att enklare avbilda det på en mer objektorienterad modell.

Okej, låt oss steg för steg beskriva hur vi bygger upp koden i ett arbetsfönster. För att inte betunga läsaren med uttröttande detaljer väljer vi att beskriva koden sekvensiellt uppifrån och ner.

Deklarationerna av de lokala variabler vi kommer att behöva ser ut som vanligt:

```
| text inStream tags outStream tag readWord doWord |
```

där text är den text med referenser som vi ska analysera, inStream en läsström på denna text, tags är en lista av märken och dess positioner i den slutliga texten, outStream en skrivström på vilken texten med referenser borttagna kommer att skrivas, tag anger 'referenstecken', readWord ett hjälpblock för att läsa nästa ord eller referens från text, samt slutligen doWord som också är ett hjälpblock som skriver på outStream respektive tags.

Efter denna deklaration tilldelas text en sträng med referenser för att vi ska kunna pröva våra algoritmer.

```
text := 'Data där vissa delar är @tag@taggade!
Dessa delar vill vi @samla@samla ihop och skriva i en speciell
@reflista@referenslista. Resten som vanlig @stream@stream.
@författare@Björn'.
```

Vi skapar en läsström, inStream, på texten.

```
inStream := text readStream.
```

Gör tags till en tom OrderedCollection.

```
tags := OrderedCollection new.
```

Strömmen outStream konstruerar vi som skrivström på en från början tom sträng. Det måste vara en skrivström för att vi ska kunna skriva på den. Att läsa hela dess innehåll går ju alltid att göra med meddelandet contents.

```
outStream := String new writeStream.
```

Tecknet som tilldelas tag anger det skiljetecken vi har använt som märke.

```
tag := $@.
```

Avsikten med blocket readWord är att läsa nästa ord eller referens. Genom att använda meddelandet peek avgörs om nästa ord är ett vanligt ord eller en tag. Ett märke börjar ju alltid med tecknet givet av tag och readWord förväntar sig att eventuellt vitt tomrum är uppätet innan anropet av blocket. Om nästa ord är en tag läser vi tom nästa tag (referenser är ju alltid inneslutna mellan två märken) och returnerar hela referensen (inklusive märken) i form av en sträng. (String with: aChar konstruerar en sträng med ett tecken, (,) konkatenerar och through: läser från aktuell position till nästa förekomst av tag inklusive)

```
readWord := [:lineStream |
    lineStream peek = tag
        ifTrue: ["Sträng från nästa tecken och efterföljande tag"
                (String with: lineStream next), (lineStream through: tag)]
        ifFalse: [lineStream upTo: Character space]].
```

Blocket doWord läser ett nytt ord, mha readWord, ser om det är ett märke och uppdaterar i så fall listan med referens-position par, annars skrivs ordet följt av ett mellanslag på utströmmen.

```
doWord :=
    [:line | | word |
    word := readWord value: line.
    word first = tag                    Är det en referens?
        ifTrue: ["Lägg till associationen, med tag som nyckel och den rena
                resultatstexten position för referensen som värde, till tags."
                tags add: (word copyFrom: 2 to: word size - 1) ->
                                (outStream position + 1)]
        ifFalse: [outStream nextPutAll: word;
                nextPut: Character space]].
```

Nu följer huvud-slingan i vilken vi läser från inStream så länge som det finns någonting kvar att läsa. För att förenkla det hela så läser vi en rad i taget itererar över raden och beroende på om det är ett styrtecken eller inte skriver resultatet direkt på outStream eller ber doWord om att göra sitt arbete på raden. Slutligen måste vi för varje rad skriva ett cr,

då upTo: läser och ställer pekaren efter nästa förekomst av Character cr men returnerar inehållet exklusive detta tecken (vilket ges till nextLine).

```
[inStream atEnd]
    whileFalse:
        [| nextLine |
        nextLine := (inStream upTo: Character cr) readStream.
        [nextLine atEnd]
            whileFalse: [nextLine peek isSeparator
                    ifTrue: [outStream nextPut: nextLine next]
                    ifFalse: [doWord value: nextLine]].
        outStream cr].
```

Nu tittar vi på resultatet.

```
outStream contents
```

⟹*Data där vissa delar är taggade!*
Dessa delar vill vi samla ihop och skriva i en speciell
referenslista. Resten som vanlig stream.
Björn

```
tags
```

⟹*OrderedCollection ('tag'->25 'samla'->55 'reflista'->92*
'stream'->125 'författare'->134)

Palindromer

Ett palindrom är en lista av element som är densamma framlänges som baklänges, eventuellt med skiljetecken borttagna. Vår uppgift är nu att skriva en unär metod isPalindrom i klassen String som ska svara på om mottagaren är ett palindrom eller ej.

Vi gör en första förenklad ansats och tillåter endast blanktecken som skiljetecken. Vidare då versaler eller gemena (stora eller små) bokstäver är signifikant vid jämförelse mellan strängar så måste vi konvertera strängen så att den enbart består av antingen den ena eller andra sortens tecken.

- Ta bort blanka

  För att ta bort alla tecken av ett visst slag (i vårt fall blanktecken) så kan vi använda meddelandet copyWithout: som konstruerar en ny sträng men med alla förekomster av tecknet som ges som argument borttagna.

- Konvertera till versaler

  För att konvertera alla tecken till versaler (alternativt gemena) så använder vi meddelandet asUppercase (alternativt asLowercase, bara man är konsekvent).

- Är strängen lika fram som baklänges?

Slutligen returnerar meddelandet reverse mottagaren baklänges och = kan som vanligt användas för att jämföra två objekt.

```
isPalindrom
    | contracted |
    contracted := (self copyWithout: Character space) asUppercase.
    ^contracted = contracted reverse
```

Nu testar vi metoden på först följande välkända palindrom

```
'Ni talar bra latin' isPalindrom
```

⇒*true*

som önskat. Men om vi däremot prövar det hela på följande sätt

```
'A man, a plan, a canal -- Panama!' isPalindrom
```

⇒*false*

detta trots att mottagaren är en palindrom!

En strategi för att klara även palindromer med skiljetecken är följande:

1 gör om mottagaren till en OrderedCollection
För att konstruera en OrderedCollection kan klassmethoden withAll: användas.

Med hjälp av meddelandet asUppercase, konvertera återigen alla tecken till versaler.

2 Ta bort alla tecken som inte är bokstäver
Om vi tar bort alla tecken som inte är bokstäver så kan vi sedan som förut direkt jämföra om det hela blir samma sak framlänges som baklänges.

Vi kan skicka meddelandet isAlphabetic till en teckenkonstant för att ta reda på om den tillhör det engelska alfabetet och om vi också vill klara av svenska tecken kan vi istället använda meddelandet isLetter. Vi väljer (självklart) det senare alternativet.

Nu kan vi återigen skriva om metoden mha av den ovan beskrivna strategin.

```
isPalindrom
    | collection |
    collection := OrderedCollection withAll: self asUppercase.
    collection copy do: [:c | c isLetter ifFalse: [collection remove: c]].
    ^collection = collection reverse
```

Denna lösning klarar även av palindromer som innehåller skiljetecken men är fortfarande inte riktigt tillfredsställande då vi bland annat behöver göra en del onödigt kopierande. Kopieringen före do: är nödvändig då vi annars får ett felavbrott som beror av att indexen för elementen inte längre stämmer under itereringen eftersom remove: är destruktiv.

Vår nästa lösning visar hur man kan använda en ström för att på ett effektivare sätt konstruera metoden.

```
isPalindrom
    "Returnerar true om self är en palindrom"
    | stream |
    stream := '' writeStream.
    self do: [:c | c isLetter ifTrue: [stream nextPut: c asUppercase]].
    ^stream contents = stream contents reverse
```

Person

Nu ska vi utvidga klassen Person från tidigare kapitel. För att enkelt kunna inspektera instanser av Person så börjar vi med att definiera en printOn:-metod.

```
printOn: aStream
    aStream nextPutAll: 'Förnamn: '.
    aStream nextPutAll: self name christianName.
    aStream nextPutAll: ' Efternamn: '.
    aStream nextPutAll: self name familyName.
    aStream cr.
    aStream nextPutAll: 'Adress: '.
    aStream cr.
    self address printOn: aStream.
    aStream cr.
    aStream nextPutAll: 'ID: '.
    self identification printOn: aStream.
```

Vi prövar de nya metoderna.

```
printOn: aStream
    (aStream tab)
        nextPutAll: self street , ' ' , self number printString; cr;
        tab; nextPutAll: self zip , ' ' , self city; cr;
        tab; nextPutAll: self country; cr;
        tab; nextPutAll: 'tel: ' , self phone
```

Vi prövar de nya metoderna.

```
| person |
person := Person christianNames: #(Yngve)
                familyName: 'Sundblad'
                identification: 5.
person
        street: 'Osquars backe'
        number: 2
        city: 'Stockholm'
        country: 'Sverige'
        zip: '10044'
        phone: '08-7906000'.
person
```

⇒

Förnamn: Yngve Efternamn: Sundblad
Adress:
Osquars backe 2
10044 Stockholm
Sverige
tel: 08-7906000
ID: 5

Vi utökar klasssen Person så att både en hemadress och en adress till ett eventuellt arbete hanteras.

```
Object subclass: #Person
    instanceVariableNames: 'name address addressWork identification '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Personregister'
```

Gamla Address får nu stå för hemadress och workAddress för adress till arbetet. För att redan nu preparera oss för framtida utvidgningar skriver vi följande metoder i Person:

KLASSMETODER

```
homeAddressClass
    "Klassen som ska användas för att lagra min hemadress"
    ^Address

workAddressClass
    "Klassen som ska användas för att lagra min adress till arbetet.
    Vi använder (än så länge) samma klass som för hemadressen"
    ^self homeAddressClass

christianNames: aCollection familyName: aString identification: uniqueID
    ^self
        christianNames: aCollection
        familyName: aString
        identification: uniqueID
        address: self homeAddressClass new
```

Avsikten med dessa metoder är att vi ska vara förberedda för troliga förändringar genom att på ett dynamiskt sätt välja klasser som hanterar adresser.

INSTANSMETODER

```
addressWork
    addressWork isNil ifTrue: [addressWork := self workAddressClass new].
    ^addressWork

workAddressClass
    "Returnera klass som hanterar adresser till arbetet."
    ^self class workAddressClass          Klassen får avgöra
```

Om vi antar att adressen till arbetet inkluderar en elektronisk postadress och ett faxnummer skulle vi kunna implementera en klass WorkAddress på följande sätt:

```
Object subclass: #WorkAddress
    instanceVariableNames: 'street number city country zip phone email fax'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Personregister'
```

Vi ser att WorkAddress till stora delar blir ekvivalent med Address. Därför kan vi ”återanvända” beskrivningar som redan finns i Address, och istället definiera WorkAddress på följande sätt:

```
Address subclass: #WorkAddress
    instanceVariableNames: 'email fax '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Personregister'
```

Åtkomstmetodena för de nya instansvariablerna skriver vi på vanligt sätt, dvs med namnen email, email:, fax och fax:.

```
street: aStreet number: aNumber city: aCity country: aCountry zip: aZip
phone: aPhone email: emailString fax: faxString
    self
        street: aStreet
        number: aNumber
        city: aCity
        country: aCountry
        zip: aZip
        phone: aPhone.
    self email: emailString.
    self fax: faxString
```

Nu får vi inte glömma att ändra Person class>>workAddressClass.

```
workAddressClass
    ^WorkAddress
```

Vi ändrar också Person>>printOn:.

```
printOn: aStream
    aStream nextPutAll: 'Förnamn: '.
    aStream nextPutAll: self name christianName.
    aStream nextPutAll: ' Efternamn: '.
    aStream nextPutAll: self name familyName.
    aStream cr.
    aStream nextPutAll: 'Adress: '.
    aStream cr.
    self address printOn: aStream.
    aStream cr.
    addressWork notNil
        ifTrue: [aStream nextPutAll: 'Arbete: '.
            aStream cr.
            self addressWork printOn: aStream.
            aStream cr].
    aStream nextPutAll: 'ID: '.
    self identification printOn: aStream
```

Slutligen definieras WorkAddress>>printOn:.

```
printOn: aStream
    super printOn: aStream.
    aStream cr; tab; nextPutAll: 'email: ' , self email printString; cr;
    tab; nextPutAll: 'Fax: ' , self fax
```

Nu testar vi genom att definiera en person.

```
| person |
person := Person christianNames: #(Björn Arne)
                familyName: 'Eiderbäck'
                identification: 2.
person address
        street: 'Gatan'
        number: 2222
        city: 'Stockholm'
        country: 'Sverige'
        zip: '10000'
        phone: '08-00123456789'.
person
```

⇒

Förnamn: Björn Efternamn: Eiderbäck
Adress:
Gatan 2222
10000 Stockholm
Sverige
tel: 08-00123456789
ID: 2

Om en adress till arbetet definieras ändras utskriften.

```
| person |
person := Person christianNames: #(Björn Arne) familyName: 'Eiderbäck'
                identification: 2.
person address
    street: 'Gatan' number: 2222 city: 'Stockholm' country: 'Sverige'
    zip: '10000' phone: '08-00123456789'.
person addressWork
    street: 'Osquars backe' number: 2 city: 'Stockholm' country: 'Sverige'
    zip: '10044' phone: '08-7906000'
    email: 'bjorne@nada.kth.se' fax: '08-7900930'.
person
```

⇒

Förnamn: Björn Efternamn: Eiderbäck
Adress:
Gatan 2222
10000 Stockholm
Sverige
tel: 08-00123456789
Arbete:
Osquars backe 2
10044 Stockholm
Sverige
tel: 08-7906000
email: bjorne@nada.kth.se
Fax: 08-7900930
ID: 2

Sammanfattning

Smalltalk

Strömmar används för att hantera läsning och skrivning av indexerbara objekt.

En ström skapas genom att direkt använda lämplig strömklass och ge det indexerbara objektet som den ska verka på som argument eller genom att skicka något av meddelandena readStream eller writeStream direkt till objektet.

Slumptal skapas från klassen Random.

Objekt använder printOn:, printString och displayString för att presentera sig själva.

Objekt lagras på strömmar med storeOn: och läses med readFrom:.

Metodik

Börja med att utveckla kod i ett arbetsfönster (eng Workspace) för att sätta in dig i problemet innan du definierar klasserna.

Många problem blir enkla att hantera och lösa med hjälp av strömmar.

Övningar

10.1 På sidan 355 konstruerade vi en klass för att konvertera en sträng av romerska siffror till dess decimala motsvarighet. Med den definition vi gjorde klarar vi endast av romerska tal som är givna på ett riktigt sätt. Ett romerskt tal ska skrivas med de mest signifikanta positiva siffrorna längst till vänster, dvs med fallande värden från vänster till höger. Så en uppenbar utvidgning av klassen vore att kontrollera att talen alltid är skrivna på detta sätt, dvs om det finns något positivt tal till höger i strängen som är större än något positivt tal längre till vänster, borde ett felmeddelande skrivas ut. Gör denna utvidgning!

10.2 Använd en ström för att tabulera ordfrekvenser i en text.

10.3 Korsreferenslista. Resultatet sorterat i bokstavsordning med radnummer där orden förekommer.

10.4 Med erfarenheterna från arbetsfönsterversionen av brevläsaren. Modellera och designa lämpliga klasser för att hantera texter med referenser.

10.5 Implementera en klass SmartStream som vid meddelandet next ger nästa alfanumeriska tecken på strömmen som dess versala mostvarighet, dvs om passera förbi icke läsbara tecken och konvertera små bokstäver till stora.

10.6 Konstruera en subklass till SmartStream som går från slutet och baklänges mot början på det objekt den verkar på.

10.7 Metoden isPalindrom på sidan 362 använder en ström. Skriv om den genom att använda två strömmar samtidigt. En som läser framlänges och en som läser baklänges.

10.8 Hur kan vi i textrefrensexemplet på sidan 357 märka ett visst ord med flera olika indexord?

10.9 Skriv kod som slumpar en tipskupong, representera kupongen som en vektor med 13 komponenter som vardera innehåller något av tecknen 1, x eller 2.

10.10 Definiera printOn: för sköldpaddorna från övning 4.9 så att de skriver ut sig på följande sätt.

```
Turtle
    (position = 10@12 direction = 90)
LogTurtle
    (position = 10@12 direction = 90 distance = 7)
LyxTurtle
    (position = 10@12 direction = 90 distance = 7 parea = 2)
```

Tänk på att utnyttja arv.

11 Filer

Detta kapitel besvarar bland annat följande frågor:

- Hur läser man från en fil?
- Hur skriver man på en fil?
- Hur hanterar man filer och kataloger?

11.1 Filhantering

Smalltalk innehåller klasser som hanterar den aktuella datorns filer på motsvarande sätt som en filhanterare i ett konventionellt filsystem gör. All denna hantering baseras väsentligen på klassen Filename med subklasser samt externa strömmar. De externa strömmarna fungerar i princip likadant som de interna strömmarna som beskrevs i föregående kapitel.

Filename

Filename är en abstrakt klass som på ett maskinoberoende sätt representerar filer och kataloger på sekundärminne. Däremot är subklasserna till Filename konkreta och plattformsberoende.

En representation av en fil eller katalog skapas genom instansiering av klassen Filename, med givet fil- eller katalognamn som argument. Vi kan tex skapa en fil med namnet fil.1 på följande sätt:

```
Filename named: 'fil.1'
```

Alternativt kan man skicka meddelandet asFilename till en sträng.

```
'fil.1' asFilename
```

När vi har skapat en referens till en fil, eller snarare instans av Filename, så kan vi göra en mängd olika operationer på den. Vi kan tex kontrollera om den finns och i så fall om den är läsbar, när den är skapad, dess storlek mm.

isWritable	är filen skrivbar?
exists	existerar filen?
delete	ta bort filen
fileSize	filens storlek
copyTo: anotherName	kopiera filen till en fil med namnet anotherName
renameTo: newName	byta namn på filen till newName
setWritable: boolean	göra filen (o)skrivbar
contentsOfEntireFile	hela innehållet av filen
edit	öppna en filebrowser på filen
dates	accessdata för filen
isDirectory	är filen en katalog?

Figur 11.1 Vanliga instansmetoder i Filename

Som exempel undersöker vi om filen MinFil existerar och i så fall returnerar vi dess storlek.

```
file := 'MinFil' asFilename.
file exists ifTrue: [file fileSize]
```

Metoderna i figur 11.1 går att använda även på kataloger. Det finns också många användbara klassmetoder.

named: aString	Skapa ny fil
currentDirectory	Aktuell katalog
defaultDirectory	Katalog varifrån relativa namn utgår
separator	Den aktuella plattformens tecken för att skilja kataloger åt
filesMatching: stringWithWildcards	Namn som matchar ett uttryck
volumes	Ger en lista med katalogerna på översta nivån i filhierarkin

Figur 11.2 Vanliga klassmetoder i Filename

Exempel: Information om katalog

I det här exemplet beskrivs hur man kan skapa en representation av katalogen som Smalltalkimagen startades från, läsa information om när katalogen skapades, hur många filer det finns i katalogen samt ta reda på vilka av dessa filer som i sin tur är kataloger.

Vi börjar som vanligt med att deklarera de temporära variabler som vi vill använda:

```
| directory contents |
```

Klassmeddelandet currentDirectory eller defaultDirectory ger oss en instans av Filename på den katalog som vi startade Smalltalk ifrån.

```
directory := Filename currentDirectory.
```

Meddelandet dates anger en fils status. Svaret ges i form av en katalog med olika typer av statusförändringar som nycklar och vektorer med datum och klockslag som värden.

```
directory dates.
```

Ger tex följande svar:

```
⇒ IdentityDictionary ( #statusChanged->#(7 June 1995 3:39:22 pm )
   #modified->#(7 June 1995 3:39:22 pm )
   #accessed->#(7 June 1995 3:39:22 pm ) )
```

Nu använder vi meddelandet directoryContents, som ger en lista (här en OrderedCollection) med namnen på de filer i katalogen och meddelandet size för att få reda på dess storlek.

```
contents := directory directoryContents.
contents size
```

```
⇒42
```

dvs det finns 42 filer i katalogen.

Slutligen går vi igenom listan contents, använder select: för att via meddelandet construct: konstruera instanser av Filename (dvs här relativt katalogen directory) och meddelandet isDirectory för att välja ut de filer som är kataloger.

```
contents select: [:aFilenameString |
    (directory construct: aFilenameString) isDirectory]
```

```
⇒#('visual' 'BIN' 'MAILS' 'SRC' 'ST.Book' 'ZSTUFF' )
```

Fortfarande har vi endast gått igenom en bråkdel av allt som Filename kan hjälpa och understödja oss med. Vi ska nu i rask följd konstruera ytterligare ett antal små exempel som visar på mer av de kraftfulla mekanismer för filhantering som Smalltalk erbjuder oss och allt på ett sätt oberoende av datortyp!

Exempel: Kataloghantering

Vi nu ge några praktiska uppgifter med svar som bla visar hur kataloger kan skapas, fås att byta namn och tas bort.

a) Konstruera en ny katalog med namnet *tmp* på översta nivån i filhierarkin. Om katalogen redan existerar ska ingen ny katalog skapas!

b) Pröva om vi har rätt att skriva på den i (a) konstruerade katalogen.

c) Konstruera en ny katalog med namnet SLASK som underkatalog till den i (a) skapade katalogen. Om denna katalog redan existerar pröva i tur och ordning med att ge den namnet SLASK.0, SLASK.1, ..., SLASK.9.

d) Gör katalogen från (c) skrivbar för alla.

e) Byt namn på SLASK till PLASK.

f) Gör så att PLASK inte är skrivbar längre, dvs tillåt inte att någon fil läggs till eller ändras i katalogen.

g) Tag bort katalogen.

Först deklarationerna:

```
| directory basicSubDirName subDirName i subDir newName |
```

a) Metoderna exists och makeDirectory används

Sedan konstruerar vi en fil med namnet **tmp** på översta nivån i filhierarkin. Toppkatalogen kan refereras genom att skapa en fil som i Unix har namnet '/', på en PC heter den istället '\' och på en Machintosh':'. I Smalltalk behöver vi inte bekymra oss om dessa egenheter utan kan helt enkelt skicka meddelandet separator till antingen klassen Filename eller en instans av densamma (meddelandet förstås av båda). Denna metod ger som svar en instans av Character med det för den aktuella plattformen specifika namnet. Vi omvandlar först tecknet till en sträng (tex mha String with: aCharacter) och slår ihop den med det önskade katalognamnet.

```
directory := Filename named: (String with: Filename separator), 'tmp'.
directory exists ifFalse: [directory makeDirectory].
```

b) Metoden isWritable

```
directory isWritable.
```

c) Modifiera en sträng och konstruera subkatalog med construct:.

I katalogen som vi skapade ovan konstruerar vi sen en instans av Filename med namnet SLASK . Om en sådan fil redan existerar konstruerar vi i tur och ordning en ny instans med sista siffra 0 tom 9 ända tills dess att ingen sådan fil redan finns i katalogen. Om räknaren (i) blir större än 9 ger vi upp!

```
basicSubDirName := 'SLASK'.
subDirName := basicSubDirName.
```

```
i := 0.
[(subDir := directory construct: subDirName) exists and: [i <= 9]]
    whileTrue: [subDirName := basicSubDirName, i printString.
                i := i + 1].
subDir exists ifFalse: [subDir makeDirectory].
```

d) Metoden makeWritable

Vi använder det unära meddelandet makeWritable som är möjligt att använda på alla plattformstyper, även om det inte alltid är (beroende av plattformsspecifika detaljer) garanterat att filen blir skrivbar för alla.

```
subDir makeWritable.
```

På vissa plattformar (tex Unix) går det att på ett mer finkornigt sätt ange läs, skriv och exekveringsrättigheter.

e) Vi byter namn med meddelandet renameTo:

Observera att subDir behåller sitt gamla namn även efter att vi har gjort namnbytet. Så vi behöver också uppdatera dess namn för att kunna lösa de efterföljande uppgifterna på enklast möjliga sätt!

```
newName := subDir head, (String with: Filename separator), 'PLASK'.
subDir renameTo: newName.
subDir named: newName.
```

Observera att vi tillåts inte att byta namn på en katalog om det redan finns en icke tom katalog med det namn vi försöker byta till. Däremot går det bra att byta namn på en katalog som innehåller filer. Det senare medför helt enkelt att filerna ”hänger med”.

f) Precis som i lösningen i e finns det ett unärt meddelande att tillgå och återigen kan vissa plattformsspecifika skillnader förekomma.

```
subDir makeUnwritable.
```

g) Meddelandet delete

```
subDir delete
```

Fler användbara metoder för Filename

Innan vi forsätter så kompletterar vi de tidigare listorna med metoder som är applicerbara på filer med ytterligare några användbara instansmetoder.

construct: str	skapar nytt filnamn där mottagaren är katalog och argumentet anger filnamn
head	mottagarens katalognamn
tail	mottagarens filnamn
directory	mottagarens katalog
asString	returnerar filnamnet
separator	returnerar katalogavdelare, /, \ eller :
directoryContents	om mottagaren är en katalog så returneras alla filer som finns i den
filesMatching: pattern	om mottagaren är en katalog så returneras alla filer med namn som uppfyller ett visst mönster (med ev jokertecknen *, #)
makeDirectory	skapa katalog
moveTo: destination	flytta fil till destination

Figur 11.3 Fler användbara metoder i Filename

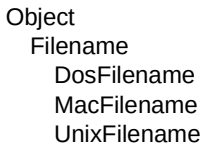
Det finns också flera sätt att konstruera externa strömmar genom att skicka meddelanden till en fil.

appendStream	skapar en ström som man kan lägga till i slutet av
newReadAppendStream	skapar eller tömmer filen som motagaren specificerar och öppnar en ExternalReadAppendStream mot den.
newReadWriteStream	skapar eller tömmer filen som motagaren specificerar och öppnar en ExternalReadWriteStream mot den.
readAppendStream	skapar eller öppnar filen som motagaren specificerar och öppnar en ExternalReadAppenStream mot den
readStream	öppnar en ExternalReadStream mot filen som mottagaren specificerar
readWriteStream	skapar eller öppnar filen som mottagaren specificerar och öppnar en ExternalReadWriteStream mot den
writeStream	skapar eller tömmer filen som mottagaren specificerar och öppnar en ExternalWriteStream mot den

Figur 11.4 Konstruktion av strömmar på filer

Filename som abstrakt klass

Klassen Filename är inte bara intressant för att den på ett maskinoberoende sätt ger användare tillgång till en mängd olika operationer att både testa och manipulera externa objekt i form av filer utan också på grund av den teknik som en konkret plattformspecifik klass väljs.



Figur 11.5 Filename, klasshierarki

Normalt sätt så instansierar man alltid från Filename, precis som vi gjort i exemplen ovan. Däremot så kommer instansen tillhöra någon av subklasserna! Exempel (på en Unixdator):

```
| file |
file := 'namn.txt' asFilename.
file class
```

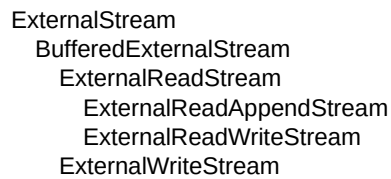
⟹ *UnixFilename*

Denna klasskonvertering sköts bakom kulisserna av Filename. Titta i systemet om du vill veta hur.

Förutom att subklasserna till Filename omdefinierar metoder definierade i Filename så finns det två utvidgningar. MacFilename innehåller även en Macintosh standarddialog för att söka/öppna filer och UNIXFilename innehåller möjligheter till att hantera rättigheter till filer och kataloger.

11.2 Externa strömmar

Externa strömmar används för att hantera själva läsningen och skrivningen av filer. De går att dela in i lässströmmar, skrivströmmar och direktströmmar, dvs strömmar som både går att läsa och skriva.



Figur 11.6 Externa strömnmar

För att stänga eller undersöka om en extern ström är stängd kan meddelanden i figur 11.7 användas.

close	stänger strömmen
closed	true om strömmen är stängd

Figur 11.7 Metoder för alla externa strömmar

De flesta metoder är definierade i BufferedExternalStream, men de omdefinieras till något vettigt där det är lämpligt och till felmeddelanden på de andra ställena.

Läsbara strömmar

ExternalReadStream är standardström för att läsa och fås genom att tex skicka meddelandet readStream till en instans av Filename. Viktiga metoder är beskrivna i figur 11.8.

next	nästa objekt på strömmen
atEnd	true om vi kommit till slutet
setToEnd	sätter positionen till slutet av strömmen
peek	nästa objekt på strömmen, utan att flytta fram positionen
reset	sätter positionen till början av strömmen
position	ger nuvarande position i strömmen
position: anInteger	sätter ny position i strömmen
upTo: anObject	strömmens innehåll fram till anObject

Figur 11.8 Metoder för läsströmmar

Skrivbara strömmar

ExternalWriteStream är standardström för att skriva och fås genom att tex skicka meddelandet writeStream till en instans av Filename.

flush	tömmer utbufferten på strömmen
next: anInteger putAll: aSequenceableCollection startingAt: startIndex	skriver anInteger antal element ur aSequenceableCollection på strömmen med början startIndex element in i aSequenceableCollection
nextStringPut: aString	skriver aString på strömmen

Figur 11.9 Metoder för skrivströmmar

nextPutAll: aCollection	skriver aCollection på strömmen

Figur 11.9 Metoder för skrivströmmar

Direktströmmar

ExternalReadWriteStream ger möjligheter att både läsa och skriva på en fil och fås genom att tex skicka meddelandet readWriteStream till en instans av Filename. ExternalReadAppendStream ger möjlighet att förlänga filen genom att först läsa och därefter lägga till i slutet. För direktströmmarna är alla ovan nämnda metoder tillämpbara.

11.3 Text på fil

Det allra enklaste man kan lagra på en fil är kanske en text. För oformatterad text kan man använda nextPutAll:

```
| file stream |
file := Filename named: 'foo'.
stream := file writeStream.
stream nextPutAll: 'Hejsan'.
stream close.
^file contentsOfEntireFile
```

⟹*'Hejsan'*

Följande exempel visar hur man kan läsa data från en fil. Här läser vi en fil med radseparerade namn.

```
| file stream names separator readBlock |
file := Filename named: 'namn.txt'.
names := OrderedCollection new.
stream := file readStream.
separator := String with: Character cr.
readBlock := [[stream atEnd]
                    whileFalse: [names add: (stream upTo: separator)]].
readBlock valueNowOrOnUnwindDo: [stream close].
^names
```

⟹ *OrderedCollection ('Olle' 'Per' 'Björn' 'Kalle' 'Hobbe')*

Har man filer indelade på andra sätt än med rader tex tabbar, kolon, komma så får man byta ut separator. Konstruktionen med valueNowOrOnUnwindDo: blir nödvändig så fort som man håller på med filer. Man vet ju aldrig vad som kan hända. Någon kan radera filen, en disk kan gå i bitar.

11.4 Objekt på fil

När man vill lagra objekt på sekundärminne använder man ofta en databas. Har man ingen databas eller om en databas är onödigt komplicerad, så finns det två olika mekanismer man kan använda, storeOn: och BinaryObjectStreamingService, BOSS. Med dessa metoder kan man lagra de flesta objekt på fil. Det är dock inte möjligt att lagra objekt som består av externa referenser. Tex går det inte att lagra en ExternalWriteStream på fil, men däremot det Filename som strömmen skapades utifrån.

Textuell lagring

Den enklaste metoden att spara objekt är med metoden storeOn: som finns definierad i klassen Object. Denna metod är tidigare beskriven i avsnitt 10.4. Vi ska här enbart ge exempel på hur man med storeOn: resp readFrom: kan lagra och läsa objekt på fil.

```
| object strm |
object := OrderedCollection new.
object add: 'Katarina'.
object add: 'Åsa'.

strm := 'objects' asFilename writeStream.
object storeOn: strm.
strm close.
```

Först skapar vi en behållare där vi lägger in strängarna *Katarina* och *Åsa*. Därefter skapas en instans av Filename genom att skicka meddelandet asFilename till strängen 'objects'. Nästa steg är att utifrån instansen av Filename skapa en skrivbar ström. Nu när vi har en instans av ExternalWriteStream så kan vi lagra behållaren där genom att skicka meddelandet storeOn: strm till behållaren. Till sist stänger vi strömmen strm med meddelandet close.

```
| object strm |
strm := 'objects' asFilename readStream.
object := Object readFrom: strm.
strm close.
object
```

⟹*OrderedCollection ('Katarina' 'Åsa')"*

För att läsa in behållaren med namn igen, så skapar vi ett objekt ur klassen Filename, men denna gång skickar vi meddelandet readStream till den så att vi får en ExternalReadStream. Nu läser vi in behållaren från filen genom att använda klassmetoden readFrom: som alla klasser förstår.

Binär lagring

Genom att använda BOSS för att lagra objekt på fil så får man snabb inläsning av objekt eftersom kompilatorn inte behöver användas. Detta ger också möjlighet att lagra cirkulära strukturer. Vidare så blir representationen kortare än med storeOn: eftersom objekten får en binär representation. BOSS ingår inte i hierarkin i figur 11.6, men de flesta omnämnda metoder fungerar likadant. Så här kan man lagra ett objekt:

```
| object boss |
object := OrderedCollection new.
object add: 'Olle'; add: 'Per'.
boss := BinaryObjectStorage onNew: 'namn.txt' asFilename writeStream.
[boss nextPutAll: object]
    valueNowOrOnUnwindDo: [boss close].
```

Vill man senare lägga till fler objekt kan man göra så här:

```
| object boss |
object := OrderedCollection new.
object add: 'Katarina'; add: 'Åsa'.
boss := BinaryObjectStorage onOld: 'namn.txt' asFilename readAppendStream.
boss setToEnd.
[boss nextPutAll: object]
    valueNowOrOnUnwindDo: [boss close].
```

För att läsa så gör man tex:

```
| object boss |
 object := OrderedCollection new.
 boss := BinaryObjectStorage onOld: 'namn.txt' asFilename readStream.
 [[boss atEnd] whileFalse:
    [object add: boss next]]
    valueNowOrOnUnwindDo: [boss close].
^object.
```

⟹ *OrderedCollection ('Olle' 'Per' 'Katarina' 'Åsa')*

11.5 Kod på fil

Även smalltalkkod kan lagras på fil. Det vanligaste sättet är att man använder **file out** i en browser. Filer skapade med **file out** kan läsas med fileIn. Exempel:

```
(Filename named: 'Filnamn.st') fileIn
```

När man gör **file out** på en klass så kommer instruktionen

```
Klassnamn initialize.
```

att läggas det sist i filen. Detta orsakar att initieringar kommer att utföras vid inläsning med fileIn, givetvis under förutsättningen att initialize är omdefinierad i klassen.

Det går också att lagra kod mha binär lagring (BOSS) men då detta är av lite mer speciell karraktär så beskriver vi inte detta närmare här utan hänvisar till systemets manualer och läsande av koden som hanterar detta.

11.6 FileConnection och IOAccessor

I det här avsnittet beskriver vi några mer avancerade tekniker som kan användas för att skapa respektive läsa och skriva på filer.

Som första exempel ska vi titta på hur man kan göra för att kopiera en fil i Smalltalk på ett effektivt sätt. Metoden vi tittar på finns definierad i klassen Filename. Metoden använder sig av den abstrakta klassen IOAccessor som på en maskinnära nivå håller i förbindelser till bla filer. Klassen hos instansen som skapas är beroende av plattformstyp, dvs Unix, PC eller Mac. Dessa objekt läser och skriver på filen buffertvis, inte tecken för tecken.

```
copyTo: destName
    "Copy the file whose name is the receiver to a file named destName."
    | buffer bufferSize sourceFile destFile amountRead |
    sourceFile := IOAccessor openFileReadOnly: self.
    destFile := IOAccessor openFileWriteOnly: destName asFilename.
    [bufferSize := sourceFile bufferSize.
    buffer := ByteArray new: bufferSize.

    [(amountRead := sourceFile readInto: buffer) > 0] whileTrue:
        [destFile writeFrom: buffer startingAt: 1 forSure: amountRead]]
            valueNowOrOnUnwindDo:
                                    [sourceFile close.
                                     destFile close]
```

Först skapas två förbindelser till filerna. Därefter frågar vi efter vilken buffertstorlek som ska användas för att få största hastighet. Bufferten skapas därnäst.

Nu är det dags att gå in i en slinga och kopiera en buffert i taget tills dess vi inte läste in något i bufferten. Metoden readInto: har två funktioner: dels att läsa in tecken till buffeten som är argument och dels att tala om hur många tecken som verkligen lästes in. Inne i slingan skriver metoden write:startingAt:forSure: ut buffertens innehåll på destinationsfilen.

Till sist ser vi till att både käll- och destinationsfil blir stängda vare sig kopieringen gick bra eller inte.

11.7 Exempel

Bankkonton lagrade på fil

I kapitel 2 konstruerade vi en klass Account. Syftet var att på ett enkelt sätt hantera bankkonton. Vi införde bland annat ett attribut för det aktuella kontots nummer, för att möjliggöra loggning av transaktioner mellan konton, men också för att förenkla ett kontos lagring på sekundärminne. Syftet med detta exempel är att visa hur vi kan använda filer för att lagra och sedan återskapa objekt av denna typ.

Vi kan direkt utnyttja att Account ärver metoder för att lagra respektive läsa objekt från strömmar och därmed också från filer.

Meddelandet storeString skickat till ett objekt visar resultatet av att skicka meddelandet storeOn: som en sträng. Så innan vi försöker lagra ett objekt på fil kan vi först pröva hur det hela kommer se ut mha storeString.

För snabb referens repeterar vi exemplet med penningtransaktioner i figur 2.4 på sidan 51. Den sista raden, den som använder meddelandet storeString, är dock ny.

```
| bigMoney expenses |
bigMoney := Account new.
bigMoney initialBalance: 100000 accountNumber: 10001.
expenses := Account new.
expenses initialBalance: 0 accountNumber: 10002.

bigMoney move: 10000 to: expenses.
expenses move: 250 to: bigMoney.
expenses storeString
```

med följande sträng som resultat:

```
'(Account basicNew instVarAt: 1 put: 9750; instVarAt: 2 put: 10002; instVarAt: 3
put: ((OrderedCollection new) add: #("INITIAL BALANS" 0 ); add: #("från: 10001"
10000 ); add: #("till: 10001" -250 ); yourself); yourself)'
```

Om vi istället vill använda en fil, tex med namnet expenses.x, så byter vi ut den sista raden i föregående exempel mot:

```
writeStream := 'expenses.x' asFilename writeStream.
expenses storeOn: writeStream.
writeStream close
```

Dvs vi konstruerar en skrivström på filen expenses.x, lagrar expenses på filen mha meddelandet storeOn: samt avslutar det hela med att stänga filen. Om ni tittar på filen så ser ni att den ser ut som resultatet från det föregående exemplet! För att återskapa det sparade kontot genom

att läsa det från filen kan vi använda en lässström istället. Tex enligt följande:

```
| readStream expenses |
readStream := 'expenses.x' asFilename readStream.
expenses := Account readFrom: readStream.
readStream close.
'Konto: ', expenses number printString, '
Balans: ', expenses balance printString, '
Transaktioner: ', expenses transactions printString
```

Vi tilldelar först den temporära variabeln readStream en lässström på filen expenses.x, läser expenses genom att uppmana klassen Account att läsa ett objekt från strömmen. Slutligen stänger vi strömmen och skriver ut lite formaterad information om expenses med följande resultat:

```
 'Konto: 10002
Balans: 9750
Transaktioner: OrderedCollection (#("INITIAL BALANS" 0 ) #("från: 10001"
10000 ) #("till: 10001" -250 ) )'
```

Vi får också direkt möjlighet att lagra flera instanser på en och samma fil, och enkelt läsa in resultatet igen vid senare tillfälle, om vi använder en Collection för att lägga in flera kontoobjekt.

```
| writeStream accounts bigMoney expenses |
accounts := OrderedCollection new.
bigMoney := Account new.
bigMoney initialBalance: 100000 accountNumber: 10001.
accounts add: bigMoney.
expenses := Account new.
expenses initialBalance: 0 accountNumber: 10002.
accounts add: expenses.
bigMoney move: 10000 to: expenses.
expenses move: 250 to: bigMoney.
writeStream := 'Accounts.x' asFilename writeStream.
writeStream store: accounts.
writeStream close
```

På filen hamnar en beskrivning av behållaren och de ingående objekten. Denna beskrivning använder inte någon speciell kunskap om objekt. Informationen formateras inte heller på något speciellt sätt. Det hela ser helt enkelt ut som följer:

```
((OrderedCollection new) add: (Account basicNew instVarAt: 1 put: 90250;
instVarAt: 2 put: 10001; instVarAt: 3 put: ((OrderedCollection new) add:
#('INITIAL BALANS' 100000 ); add: #('till: 10002' -10000 ); add: #('från: 10002'
250 ); yourself); yourself); add: (Account basicNew instVarAt: 1 put: 9750;
instVarAt: 2 put: 10002; instVarAt: 3 put: ((OrderedCollection new) add:
#('INITIAL BALANS' 0 ); add: #('från: 10001' 10000 ); add: #('till: 10001' -250 );
yourself); yourself); yourself)
```

Nästa kodavsnitt använder sedan filen för att återskapa behållaren, accountsCollection, inklusive de tidigare ingående kontoobjekten.

```
| accountsCollection file |
file := 'Accounts.x' asFilename readStream.
file exists
    ifTrue: [| readStream |
        readStream := file readStream.
        accountsCollection := Collection readFrom: readStream.
        readStream close]
    ifFalse: [file error: 'Finns ej!']
```

Om vi däremot vill lagra det hela på ett tydligare, kompaktare eller mer systemoberoende sätt kan vi skriva om instansmetoden storeOn: samt klassmetoden readFrom: som beskrevs i avsnitt 10.4.

Cirkulär lista

Nu ska vi göra ett par utvidgningar till klassen DoubleLinkedCircularChain från avsnitt 8.1. Som vi tidigare nämnt så går det inte att lagra cirkulära strukturer mha storeOn: utan att ingående klasser själva definierar om denna metod och hanterar cirkuläriteten.

INSTANSMETODER

```
storeOn: aStream
    "Skapa en presentation av mig och mina element på den givna strömmen"
    | valueArray |
    valueArray := OrderedCollection new.
    self do: [:aLink | valueArray add: aLink value].
    aStream store: self class.
    aStream nextPutAll: ' fromArray: '.
    valueArray asArray storeOn: aStream
```

KLASSMETODER

```
fromArray: anArray
    "Jag återskapar en instans av klassen med element givna av med värden
    givna av vektorn anArray"
    | list element |
    list := DoubleLinkedCircularChain value: anArray first.
    element := list.
    (2 to: anArray size)
    do: [:val | element := element add: val].
    ^list
```

Vi prövar det hela.

```
| list element store |
list := DoubleLinkedCircularChain value: 1.
element := list.
(2 to: 5) do: [:i | element := element add: i].
store := list storeString.
```

⟹*'DoubleLinkedCircularChain fromArray: #(1 2 3 4 5)'*

```
list class readFromString: store
```

⟹*{1 2 3 4 5 }*

Papperskorg

I moderna direktmanipulativa gränssnitt tar man aldrig bort en fil då man ber om att den ska raderas från en viss katalog. Istället brukar en fil som är ”borttagen” placeras i en papperskorg med möjlighet för användaren att ångra borttaget. En papperskorg töms inte heller förrän användaren särskilt ber om det. Om man däremot ”placerar” en ny fil med samma namn som en existerande fil i en papperskorg brukar den redan befintliga filen få ett nytt namn som indikerar att den är en äldre dubblett.

I det här exemplet ska vi konstruera en klass som hanterar en enkel papperskorg av det ovan beskrivna slaget. Vi använder en speciell katalog för vår hantering och antar att kommunikationen med katalogen i huvudak sker med hjälp av en instans av vår klass.

```
Object subclass: #Trash
    instanceVariableNames: ''
    classVariableNames: 'BackupExtension TrashDirectory '
    poolDictionaries: ''
    category: 'bok-1-filer'
```

INSTANSMETODER

constants

```
trashDirectory
    "Returnerar papperskorgens katalog"
    ^self class trashDirectory
```

accessing externals

```
delete: aFilenameOrString
    "Ta bort filen som ges som argument"
    self deleteFile: aFilenameOrString asFilename
```

accessing internals

```
contents
    "Hela papperskorgens innehåll"
    ^self trashDirectory directoryContents
```

```
empty
    "Töm papperskorgen"
    self emptyAll: (self contents collect: [:aName | self construct: aName])

emptyFilesMatching: string
    "Ta bort alla filer i papperskorgen som matchar sökmönstret i string"
    self emptyAll: (self filesMatching: string)

filesMatching: string
    "Filerna i papperskorgen som matchar sökmönstret i string"
    ^self trashDirectory filesMatching: string
```

private

```
backupOldFileNamed: aFile
    "Om en fil med samma namn som aFile redan finns i papperskorgen byt namn
    på den genom att lägga på ett suffix givet av klassmetoden backupExtension"
    | nameString oldFile |
    nameString := aFile asString.
    oldFile := self construct: nameString.
    oldFile exists
        ifTrue:
            ["Det finns fil med samma namn"
            | extension backupFile extraSuffix count |
            extraSuffix := ''.
            count := 0.
            "Konstruera unikt filnamn"
            [extension := self class backupExtension , extraSuffix.
            (backupFile := self construct: nameString , extension) exists]
                whileTrue:
                    ["Om fil med aktuellt suffix finns så skapas nytt suffix"
                    extraSuffix := count printString.
                    count := count + 1].
            oldFile renameTo: backupFile asString]

construct: aName
    "Konstruera en instans av Filename som representerar en fil med namnet
    aName i papperskorgen"
    ^self trashDirectory construct: aName

deleteFile: aFile
    "Flytta aFile till papperskorgen"
    self backupOldFileNamed: aFile.
    aFile renameTo: (self construct: aFile tail).

emptyAll: files
    "Ta bort filerna files"
    files do: [:aFile | aFile asFilename delete]
```

KLASSMETODER

constant settings

```
backupExtension
    "Suffix som läggs till filer som flyttas till papperskorgen om det redan finns en
    fil med samma namn där"
    ^BackupExtension
```

```
defaultTrashDirectory
    "Skönsvärdet för papperskorgen som instans av Filename"
    ^self defaultTrashDirectoryName asFilename

defaultTrashDirectoryName
    "Placering av papperskorgen om inget annat anges"
    ^'TRASH'

trashDirectory
    "Papperskorgens katalog"
    ^TrashDirectory

class initialization

initialize
    "Initiering av klassen"
    self initializeBackupExtension

initializeBackupExtension
    "Initiera suffixet för backupExtension"
    BackupExtension := '.back'

trashDirectory: aFilenameOrString
    "Ange var papperskorgen finns"
    TrashDirectory := aFilenameOrString asFilename.
    TrashDirectory exists ifFalse: [TrashDirectory makeDirectory].

instance creation

named: aFilenameOrString
    "Skapa en papperskorgskatalog"
    self trashDirectory: aFilenameOrString.
    ^self new

new
    "Öppna papperskorgen med placering given av aktuell inställning"
    (self trashDirectory isNil or: [self trashDirectory exists not])
        ifTrue: [self trashDirectory: self defaultTrashDirectory].
    ^super new
```

Låt oss anta att vi har filer med namn **`1.tmp, 2.tmp, ... , 10.tmp`**. Sedan utför vi:

```
| trash |
trash := Trash new.
trash delete: '2.tmp'.
trash delete: '3.tmp'.
trash contents
```

⟹*#('2.tmp' '3.tmp')*

Vi skapar en ny fil men namnet **`2.tmp`** och tar genast bort den (dvs den flyttas till papperskorgen).

```
'2.tmp' asFilename writeStream close.
trash delete: '2.tmp'.
trash contents
```

Vilka filer i papperskorgen matchar ett visst mönster?

```
⟹ #('2.tmp.back' '3.tmp' '2.tmp')
```

```
trash filesMatching: '*.back'
```

```
⟹OrderedCollection ('TRASH/2.tmp.back')
```

Vi tar bort några filer som matchar ett mönster.

```
trash emptyFilesMatching: '2.###'.
trash contents
```

```
⟹#('2.tmp.back' '3.tmp')
```

Syntaxanalys av elektronisk post

Idag sprids användningen av elektronisk post väldigt snabbt. Vanligen inleds breven som skickas mellan de olika användarna med ett inledande (brev-)huvud som anger ämne, avsändare, mottagare (eventuellt flera) och en del annan information. Efter detta huvud kommer själva brevets innehåll, brevkroppen, i form av en enda textsträng.

I ett av de mer spridda brevsystemen består ett brev alltid av ett brevhuvud med ett antal olika rubriker följda av ämnesinformation knuten till den aktuella rubriken. Huvud och kropp åtskiljs med en textrad som endast består av en radframmatning.

```
Object subclass: #BokMail
    instanceVariableNames: 'header body '
    classVariableNames: 'HeaderBodySeparator MailTag NewLine TagEnd '
    poolDictionaries: ''
    category: 'Brevhantering'
```

KLASSMETODER

```
initializeConstants
    "Initiera användbara konstanter"
    NewLine := Character cr.
    MailTag := (String with: NewLine), 'From '.
    TagEnd := $:.
    HeaderBodySeparator := String with: NewLine with: NewLine

initialize
    "Initiera klassen"
    self initializeConstants

readFromStream: aStream
    "Läs nästa brev från strömmen"
    aStream atEnd ifTrue: [self error: 'Strömmen slut!'].
    ^self new readFromStream: aStream
```

INSTANSMETODER

```
readFromStream: aStream
    "Skapar en katalog för brevhuvudets märkord samt för brevhuvud och
    brevkropp"
    header := Dictionary new.
    self readHeaderFrom: aStream.
    self readBodyFrom: aStream
```

För att inte inveckla oss i detaljer väljer vi att i följande metod först ta reda på slutet av huvudet innan vi gör någon vidare behandling av det hela. Detta innebär ju att vi först ”läser” egenom hela brevet fram till slutet för att sedan åter starta från början vilket inte ur prestandahänseende är det mest effektiva. Men som sagt för att begränsa och förenkla koden väljer vi ändå att göra på detta sätt.

```
readHeaderFrom: aStream
    "Läs brevhuvudet, ta reda på nyckelord och text och stoppa in detta i
    brevhuvudkatalogen"
    | headerStream tag contents |
    headerStream := (aStream throughAll: HeaderBodySeparator) readStream.
    "Första märkordet är speciellt och används för internt bruk!"
    tag := 'START-FROM-TAG'.
    contents := ''.
    [headerStream atEnd]
        whileFalse:
            [| nextLine |
            nextLine := self divideLine: (aStream upTo: NewLine).
            nextLine key isNil
                ifTrue: [contents := contents , nextLine value]
                ifFalse:
                    [self atTag: tag put: contents.
                    tag := nextLine key.
                    contents := nextLine value]].
    self atTag: tag put: contents
```

Nu ska brevets kropp bestå av det som följer på strömmen upp till nästa MailTag. Om inte någon sådan tag finns så kommer innehållet på resten av strömmen returneras!

```
readBodyFrom: aStream
    "Läs brevkroppen"
    body := aStream upToAll: MailTag.
    "Vi förbereder också för läsning av nytt brev!"
    aStream skipSeparators

atTag: tag put: contents
    "Stoppa in contents vid angiven tag i header-katalogen"
    (header includesKey: tag)
        ifFalse: [header at: tag put: OrderedCollection new].
    (header at: tag) add: contents
```

Följande metod undersöker om aktuell rad innehåller ett märke på giltigt format. Om så är fallet returneras den som en association (nyckel-värde-par), där märket är nyckel och resten av raden blir dess värde. Om vi däremot finner att raden inte inleds med nyckelord så kontrueras en association bestående av nil som nyckel och hela raden som värde. Om nyckeln inte är nil så använder vi metoden skipSeparator för att ta bort skiljetecken som inleder ”värdesträngen”.

```
divideLine: aLine
    "Analysera raden och returnera en association med eventuell tag som nyckel
    och resten av innehållet som värde. Om raden inte innehåller tag så blir
    nyckeln nil. Inledande blanktecken tas också bort från innehållet."
    | lineStream tagStream containsTag stop |
    lineStream := aLine readStream.
    tagStream := '' writeStream.
    containsTag := false.
    stop := false.
    [stop]
    whileFalse:
        [| next |
        next := lineStream next.
        next = TagEnd
            ifTrue: [stop := containsTag := true]
            ifFalse: [(next isNil or: [next isSeparator])
                        ifTrue: [stop := true]
                        ifFalse: [tagStream nextPut: next]]].
    ^containsTag
        ifTrue: [tagStream contents asUppercase ->
                    (lineStream skipSeparators; upToEnd)]
        ifFalse: [nil -> aLine]


Object subclass: #BokReadSpool
    instanceVariableNames: 'mails '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Brevhantering'

INSTANSMETODER

initialize
    "mails initieras. Senare kommer brev som läses från brevströmmen placeras
    i den"
    mails := OrderedCollection new
```

I nästa metod går vi igenom hela strömmen och lägger upp respektive brev som en komponent i en ordnad mängd. Observera att vi låter den aktuella klassen som hanterar ett brev själv läsa information från strömmen. Detta är analogt med användandet av klassmetoden readFrom: med vars hjälp varje klass vet hur den ska återskapa en instans av sig själv från en given ström.

```
readFromStream: aStream
    "Läs alla brev på strömmen"
    [aStream atEnd]
        whileFalse: [mails add: (self mailClass readFromStream: aStream)]

mailClass
    "Klass som ska användas för att läsa ett enskilt brev"
    ^BokMail

mails
    ^mails
```

KLASSMETODER

```
new
    ^super new initialize

readFromStream: aStream
    ^self new readFromStream: aStream
```

För att på ett smidigt sätt få ett gränssnitt mot spoolfilen skriver vi också följande klassmetod (metoderna spoolDirectory och spoolFile återkommer vi till nedan):

```
readSpool
    "Skapa en lässström på den fil som innehåller alla nyinkomna brev"
    | spoolFile |
    spoolFile := (self spoolDirectory , self spoolFile) asFilename.
    spoolFile exists ifFalse: [self error: 'Ingen fil med namnet ' , spoolFile asString
    , ' finns!'].
    self readFromStream: spoolFile readStream
```

För de av er som arbetar i en omgivning där en CEnvironment existerar (laddas in från **utils**) och varje användare har en egen spoolkatalog kan följande bekväma (automatiskt anpassningsbara till aktuell användare) metoder för att definiera spoolfilen användas:

```
spoolDirectory
    "Var är brevspoolern (i det här fallet på vanlig plats i UNIX)"
    ^'/var/spool/mail/'
```

som framgår har vi här valt att definiera filnamnet på Unix-format. En kanske snyggare lösning vore att istället använda en klassvariabel med plattformsspecifik bindning till filnamnet.

```
spoolFile
    "Returnera den aktuella personens användarnamn"
    ^CEnvironment getenv: 'USER'
```

om inte denna metod fungerar på den aktuella plattformen kan tex det aktuella filnamnet explicit returneras från metoden.

Filsökare

En ofta efterfrågad tjänst är en som hjälper användaren att hitta filer i olika delar av den många gånger gigantiska mängden filer som finns på disken. För att göra denna sökning vill användaren kunna ange olika typer av restriktioner på sökningen tex hur namnet ska se ut, med eventuella jokertecken (eng: wildcards), typ av filer, storlekar, datum och kanske en hel del mer.

Vi skissar några detaljlösningar i en filsökare, men lämnar som övning att konstruera metoder och klasser av det hela.

a) ange startkatalog och namn på eftersökta filer med eventuella jokertecken

```
| startDirectory filePattern files |
startDirectory := Filename defaultDirectory.
filePattern := '*.st'.
files := startDirectory filesMatching: filePattern.
```

b) ge minimumstorlek på filerna

```
files select: [:aFile | aFile asFilename fileSize > 10000]
```

c) endast filer som inte ändrats idag

```
files select: [:aFile | (aFile asFilename dates at: #modified) first < Date today]
```

Vi anger att det är ändringstillfället vi är intresserade av. Från dates får man en Array där första position är datum och andra är klockslag

d) sök efter första förekomst och inte efter alla

```
files detect: [:aFile | (aFile asFilename dates at: #modified) first < Date today]
```

e) sökning efter filer som inte uppfyller vissa villkor
som ovan fast med reject: eller negationer i sökblocken

Sammanfattning

Allmännt

I det här kapitlet har vi visat hur Filename och externa strömmar kan användas för att hantera filer och kataloger. Filer och externa strömmar är plattformsoberoende. Gränssnitten mot strömmar i primärminnet, se kapitel 11, och strömmar på filer är förutom vissa detaljer ekvivalenta.

Programmeringsstil

Använd den abstrakta klassen Filename för att hantera filer på ett plattformsoberoende sätt.

Smalltalk

Filename används för att hantera filer och kataloger.

Objekt lagras mha BinaryObjectStreamingService, BOSS eller storeOn:.

Strömmar på filer skapas från Filename. Finns läsströmmar, skrivströmmar, direktströmmar och strömmar som läser fritt men lägger till i slutet.

FileConnection och IOAccessor kan användas vid blockvis läsning och skrivning.

Övningar

11.1 Konstruera en instans av fil med namnet MinFil genom att

a) använda lämpligt klassmeddelande i klassen Filename

b) utnyttja en sträng och lämpligt unärt meddelande.

11.2 På sidan 389 konstruerade vi en enkel syntaxanalysator för att läsa elektronisk post från en ström av brev. Klassen BokMail saknar dock åtkomstmetoder och skulle med enkla medel kunna ge sina instanser en lite trevligare textuell representation. Skriv följande i klassen BokMail:

a) En metod body som returnerar brevets kropp!

b) Inspektorerna subject, from, to, cc, date, id och inReplyTo som returnerar innehållet i katalogen header för nycklarna 'SUBJECT, 'FROM', 'TO', 'CC', 'DATE', 'MESSAGE-ID' respektive 'IN-REPLY-TO'. Om aktuell nyckel inte finns i katalogen ska nil returneras från respektive metod.

c) För ett brev ska representera sig lite snyggare så kan vi definiera om metoden printOn:. Skriv om denna metod så att ett brev presenteras med nyckelord med värden ur katalogen följt av brevkroppen. Låt fortfarande värdena vara instanser av OrderedCollection.

11.3 I övning 11.2 ”snyggade” vi upp brevklasserna på sidan 389 en aning. En brist som klassen BokMail fortfarande har är att representationen av instanser ur denna klass fortfarande presenterar sina värden som instanser av OrderedCollection. Vidare ger accessfunktionerna nil som svar om en viss nyckel ej finns.

a) Konstruera en subklass till BokMail där alla de i 11.2 b) konstruerade metoderna omdefinieras i den nya klassen, fast fortfarande med utnyttjan av de i superklassen existerande metoderna, så att de istället returnerar det första elementet ur respektive värdemängd. Om inte aktuell nyckel existerar ska vidare den tomma strängen, istälet för som tidigare nil, returneras.

b) Skriv också en ny subklass till BokReadSpool. Se till att den, istället för att använda BokMail, automatiskt använder den i (a) konstruerade klassen, (LEDNING: en metod måste definieras om i den nya klassen)

11.4 Skriv filsökaren från sidan 393 som en klass.

a) Skriv filsökaren så att den söker ner i underkataloger

b) Skriv filsökaren så att man kan fortsätta leta efter fler filer som passar med vilkoren om man valde att stanna vid första fynd av fil.

11.5 I en textfil finns instoppade tal. Läs igenom filen och summera alla dessa tal.

11.6 Använd en direktström för att byta ut alla små bokstäver mot stora i en textfil.

11.7 Ofta får man filer där det svenska tecknen åääöÅÄÖ är ersatta med }{|][\. Använd en direktfil för att konvertera alla tecken till svenska tecken.

11.8 Skriv en klass WordFinder som kopplad till en textfil kan söka reda på ord i filen som innehåller en sträng. Strängen skall kunna ges med jokertecken (# och *) för att indikera enstaka eller multipla obestämda tecken.

11.9 Skriv en klass BinaryWordFinder som given en textfil med ord sorterade i bokstavsordning, med blanktecken mellan orden:

a) Den ska kunna svara på om ett givet ord finns i filen
b) Använd ett Dictionary eller liknande för att ”komma ihåg” vilka ord som har sökts. Så vid kontroll av om ordet finns ska först detta minne konsulteras.

11.10 Skriv en klass för stavningskontroll av textfiler genom att utnyttja klassen som skrevs i övning 11.8. Alla ord i texten som inte finns med i ordlistan ska bli en association med ord->position i en lista som sedan returneras som resultat av kontrollen.

11.11 **`Stofil`** innehåller namnen på de främsta travstona genom tiderna, ett namn per rad i bokstavsordning. Nu ber vi dej att samsortera **`Stofil`** med en liknande **`Hingstfil`** så att man får en **`Kusefil`**.

Stofil

Ada Lovelace
Askungen
Belle
Betty Boop

Hingstfil

Admiral von Schneider
Al Bundy
Bart Simpson
Ego Boy

Kusefil

Ada Lovelace
Admiral von Schneider
Al Bundy
Askungen
Bart Simpson
Belle
Betty Boop
Ego Boy

11.12 Posterna i filen **`Sjukfil`** är av typen sjukposttyp och har bla instansvariblerna pNr och inkomst (till sjukkassan uppgiven inkomst). I filen **`Dagisfil`** är posterna av typen dagisposttyp och också i den finns instansvariablerna pNr och inkomst (inkomst som uppgivits för dagis). Båda filerna är ordnade efter pNr. Antag att alla i dagisfilen finns även i sjukfilen. Posterna kan i det här fallet betraktas som värden för instansvari-

ablerna i klasserna SjukPerson resp DagisPerson, de gemensamma instansvariablerna är givetvis ärvda från Person.

Gör en samkörning med utskrift av personer som finns i båda registren och har minst 20% lägre inkomst i dagisfilen än i sjukfilen. Fundera samtidigt över om du ska meddela försäkringskassan eller de berörda personerna resultatet.

11.13 Skriv kod som beräknar ordfrekvenser i en infil. Använd Dictionary eller Bag.

11.14 Sköldpaddsslussen i övning 7.3 stänger för natten och sköldpaddorna behöver sova över i en fil. Skriv kod som sparar sköldpaddskön på fil och visa hur man läser in filen igen.

12 Avlusning och felhantering

Detta kapitel besvarar bland annat följande frågor:

- Hur gör man spårutskrifter?
- Hur kan man använda ett inspektionsfönster?
- Hur använder man avlusaren?
- Hur ordnar man brytpunkter?
- Hur hanterar man projekt?
- Vad är en ChangeList?
- Hur återskapar man kod efter en krasch?

Vi har sparat en mer strukturerad beskrivning av avlusare och felhantering tills nu. Orsakerna är många men den främsta är att man bör ha en viss kännedom om systemet och utvecklat vissa programmeringsfärdigheter i språket innan man till fullo kan uppskatta och ta till sig en beskrivning av detta.

12.1 Avlusningtekniker

Fel i program är av olika typer och yttrar sig på olika sätt. Beroende på hur felet yttrar sig så behöver man olika metoder för att leta sig fram till felet. Vi ska här beskriva hur man kan använda *spårutskrifter*, *objektinspektion* och *brytpunkter* för att följa program och dataflöde.

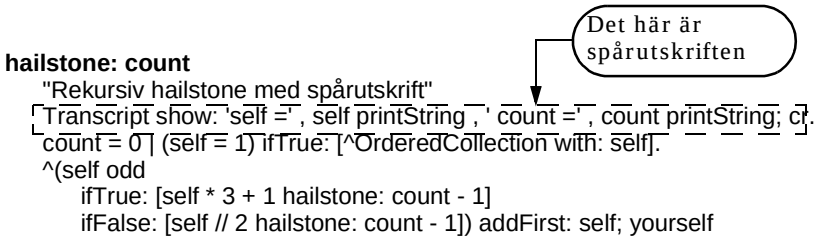
Utseendet och vissa detaljer skiljer mellan de olika smalltalkversionerna, men principerna är desamma. Här är fönstren från VisualWorks och både från Macintosh och X. För en närmare beskrivning av gränssnittet till avlusarna i andra miljöer, se respektive bok ”Smalltalkmiljön i ...”.

Spårutskrifter

En enkel och mycket användbar metod för att finna fel i program är att vid vissa väl valda punkter i programmet skriva ut något, dvs sk

spårutskrifter används. Dessa fyller i huvudsak två uppgifter nämligen 1) att tala om att programmet just utför ett visst avsnitt 2) visa variablers värden och tillstånd vid vissa tidpunkter genom att skriva ut dess värden. I Smalltalk är det enklast att skriva spårutskriften i utskriftsfönstret.

Som exempel inför vi spårutskrifter i en rekursiv variant av Hailstonefunktionen, hailstone:, från sidan 32.




```
hailstone: count
    "Rekursiv hailstone med spårutskrift"
  Transcript show: 'self =', self printString , ' count =', count printString; cr.
   count = 0 | (self = 1) ifTrue: [^OrderedCollection with: self].
   ^(self odd
        ifTrue: [self * 3 + 1 hailstone: count - 1]
        ifFalse: [self // 2 hailstone: count - 1]) addFirst: self; yourself
```

Den här varianten av hailstone: returnerar en OrderedCollection som består av alla tal ur hailstoneserien, till skillnad mot originalet i avsnitt 1.5 som skrev ut talserien i utskriftsfönstret.

Först kontrollerar metoden om något av avbrottsvilkoren är uppfyllda, dvs om nuvarande tal i talserien är ett eller om vi har nått maxlängden av serien. Därefter kontrollerar den om self är udda och i så fall anropas hailstone: med self*3 +1, i annat fall med self / 2. Som resultat från funktionen lämnas en OrderedCollection där self läggs till som första tal. På det sättet så kommer varje rekursivt anrop att lägga till det egna talet i början av listan.

Om vi nu prövar:

```
5 hailstone: 10
```

⟹*OrderedCollection (5 16 8 4 2 1)*

I utskriftsfönstret blir utskriften

```
self =5 count =10
self =16 count =9
self =8 count =8
self =4 count =7
self =2 count =6
self =1 count =5
```

Vi ser att self förändras som den ska och att count minskar med ett som det är tänkt. Resultatet av metodanropet är en OrderedCollection som innehåller hela talserien. Som vi kan se från spårutskriften så stämmer de överens och därmed verkar programmet fungera som det var tänkt.

Spårmetod i Object

Eftersom man ofta har användning för spårutskrifter så vore det bra om varje objekt enkelt kunde anropa en metod som sköter alla detaljer i spårutskriften. Genom att definiera spårutskriftsfunktionen i klassen Object så kan spårmeddelandet skickas till alla objekt. Ett exempel på implementation av en sådan metod med utmatning i utskriftsfönstret är:

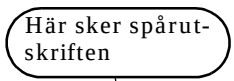
```
Object>>trace
    "ger en utskrift av mig i Transcriptfönstret"
    Transcript show: '***TRACE**** ', self printString ; cr
```

Eftersom trace är definierat på detta sätt kan vi överallt, i alla uttryck, inflika meddelandet trace utan att påverka resultatet av tidigare eller efterföljande meddelanden.

```
(2 * Float pi) trace cos
```

Uttrycket ovan kommer skriva ut ***TRACE*** 6.28 i utskriftsfönstret och därefter beräkna cosinus av detta delresultat.

Vi kan också lägga till trace i hailstone: för att följa hur resultatlistan byggs upp.




```
hailstone: count
    "Rekursiv hailstone med spårutskrift"
    count = 0 | (self = 1) ifTrue: [^OrderedCollection with: self].
    ^(self odd
        ifTrue: [self * 3 + 1 hailstone: count - 1]
        ifFalse: [self // 2 hailstone: count - 1]) addFirst: self; yourself; trace
```

om vi nu anropar med:

```
5 hailstone: 10
```

så blir spårutskriften:

```
***TRACE***OrderedCollection (2 1)
***TRACE***OrderedCollection (4 2 1)
***TRACE***OrderedCollection (8 4 2 1)
***TRACE***OrderedCollection (16 8 4 2 1)
***TRACE***OrderedCollection (5 16 8 4 2 1)
```

Men vi vill också skilja olika spårutskrifter ifrån varandra. Det vore bra om det tex gick att identifiera varifrån anropet till trace skedde. Ett sätt att göra det är att förse spårmetoden trace med ett argument. För detta ändamål definierar vi en ny spårmetod '!' och får då:

```
Object>> ! str
    Transcript show: '***TRACE*** ', self printString, ' in ', str ; cr
```

Motsvarande anrop till spårrutinen '!' blir nu

```
#(4 7 1) asOrderedCollection ! 'trace exempel' add: 1
```

En anledning till att använda utropstecken (!) som meddelandenamn, och inte tex trace:, är att om den används i kombination med nyckelordsmeddelanden behövs inga extra parenteser, då binära meddelanden har högre prioritet än nyckelordsmeddelanden. Men vi måste ändå vara försiktiga vid spårutskrifter med (!) om binära eller unära meddelanden ingår i de uttryck vi vill spåra.

Ofta vill vi veta varifrån spårutskriften kommer så därför ska vi skriva om spårutskriftsmetoden trace så att också anropspunkten identifieras. I denna version kommer vi att utnyttja variabeln thisContext som beskriver exekveringsomgivning och metodanropskedjan. Variabeln används av systemet i bla avlusaren och vid avbrottshantering (exception handling) vilket beskrivs senare i kapitlet. Med hjälp av thisContext kan både mottagaren av meddelandet och metodnamn skrivas ut, vilket vi utnyttjar.

```
trace
    Transcript show: '***Trace***' , self printString , '  in  ' ,
        thisContext sender receiver printString , '>>' ,
        thisContext sender selector; cr
```

Med anropet (2 hailstone: 20) trace kommer utskriften bli

```
***Trace***OrderedCollection (2 1)  in  2>>hailstone:
***Trace***OrderedCollection (4 2 1)  in  4>>hailstone:
***Trace***OrderedCollection (8 4 2 1)  in  8>>hailstone:
***Trace***OrderedCollection (16 8 4 2 1)  in  16>>hailstone:
***Trace***OrderedCollection (5 16 8 4 2 1)  in  5>>hailstone:
```

Det vill säga nu presenteras mottagaren av meddelandet trace, anropande objekt och den metod från vilket spårutskriften görs, dvs hailstone:.

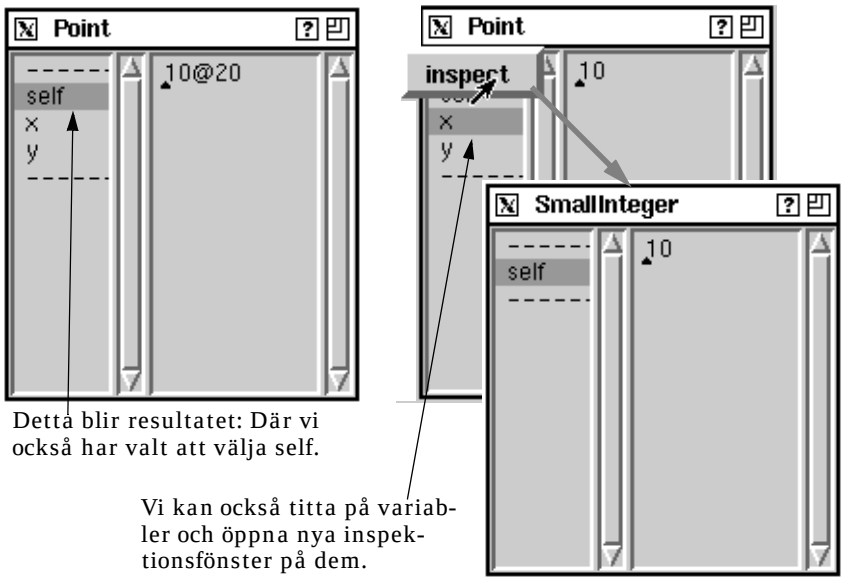
12.2 Inspektionsfönster

Mycket användbart vid utveckling av kod är alla objekts möjligheter att visa sig i ett inspektionsfönster. Ett inspektionsfönster öppnas genom att meddelandet inspect skickas till ett objekt. När detta meddelande utförs så görs först motsvarande **do it** på utvalda Smalltalk-satser och sedan öppnas fönstret på resultatet av det hela.

Alternativt kan menyalternativet **inspect** användas, vanligen från operationsmenyn. Detta innebär att systemet skickar meddelandet inspect till angivet objekt.

Om vi tex vill titta på hur en punkt är uppbyggd så kan vi först skriva smalltalksatser för att generera en punkt och sedan skicka meddelandet inspect till resultatet (se figur 12.1 nedan).

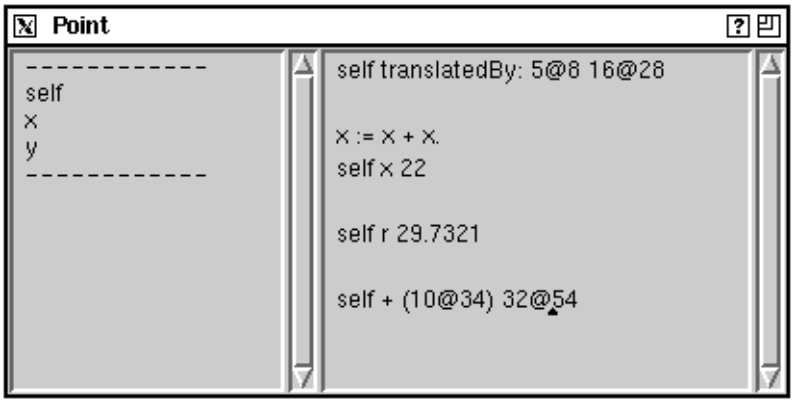
```
(Point x: 10 y: 20) inspect
```



Figur 12.1 Inspektion av en instans av klassen Point

Som framgår finns det en speciell meny i listan av variabler, från vilken man tex kan öppna nya inspektionsfönster. Den högra delen av fönstret är ett vanligt textfönster, med en meny motsvarande den i ett arbetsfönster. I princip kan man göra allt som går att göra i ett arbetsfönster här, plus lite till. Här arbetar man i en omgivning av det inspekterade objektet, vilket innebär att alla instansvariabler och objektet själv kan användas i Smalltalkuttryck. Variablerna kan ändras genom tilldelning eller genom att välja ut en variabel ändra värdet i textfönstret och välja menyalternativet **inspect**. För att referera det inspekterade objektet används pseudovariabeln self.

I figur 12.2 nedan har vi använt inspektfönstret för att pröva några metoder som definieras av klassen samt tilldela instansvariabeln x ett nytt värde.



Figur 12.2 Testa och koda i ett inspektionsfönster

Varje klass kan definiera utformning av ett inspektionsfönster. Därmed kan presentationen skilja sig lite beroende av vilken typ av objekt som undersöks.

Exempelvis så använder vissa Collection-klasser egna typer av inspektionsfönster, där inte varje detalj i klassen visas. Om alla detaljer önskas kan istället meddelandet basicInspect användas.

De speciella variablerna kan tex vara sådana som inte direkt har att göra med ett visst objekts utseende eller beteende. I en del klasser av behållartyp finns tex delar som är allokerade men kanske inte utnyttjas ännu samt variabler som internt beskriver objektets struktur. Ofta har de speciella inspektionsfönstrena även anpassade menyer.

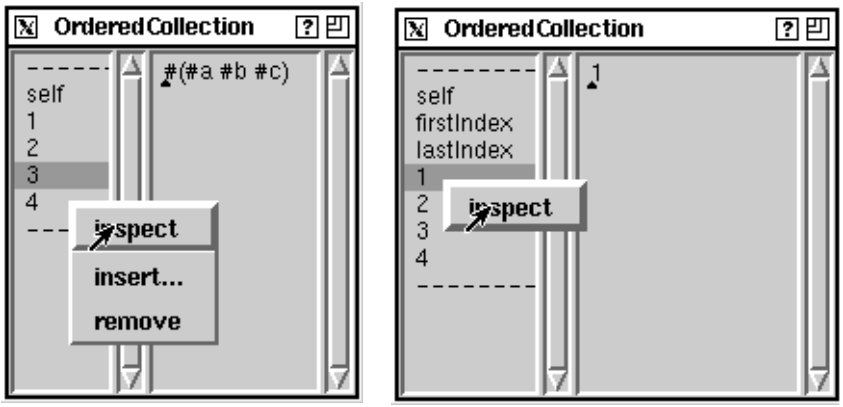
Som exempel kan vi illustrera skillnaden mellan inspect och basicInspect genom att öppna en av varje sort på en instans av OrderedCollection. Här ser vi också hur anonyma variabler i objekt av indicerbar typ presenteras, dvs med sifferkod för index.

I det aktuella fallet ser vi att för närvarande är inget extra utrymme reserverat för ännu ej använda platser.

```
#(1 2 #(a b c) 3) asOrderedCollection inspect
```

respektive

```
#(1 2 #(a b c) 3) asOrderedCollection basicInspect
```



Figur 12.3 Till vänster resultatet av inspect och till höger av basicInspect

Avlusning med inspektionsfönster

Vi har sett hur spårutskrifter kan användas för att skriva ut data och följa hur de ändras. Därigenom kan fel enkelt upptäckas och korrigeras. En svårighet med denna metod är att avgöra hur mycket information som ska presenteras. Tillräckligt mycket måste skrivas ut för att ge en klar bild av felet men samtidigt får inte för mycket information presenteras så att helheten förloras.

Om istället ett inspektionsfönster används på något centralt objekt i tillämpningen kan sedan detta användas för att undersöka relationer, instansvariabler och pröva metoder. Från detta inspektionsfönster går det också att vid behov öppna nya inspektionsfönster på det aktuella objektets olika delar eller objekt skapade genom meddelandesändning till det.

Det går också bra att ha inspektionsfönster öppna samtidigt som tillämpningen exekverar. Därigenom går det att när som helst undersöka de olika objekten för att försäkra sig om att dem uppför sig på önskat sätt.

12.3 Avlusaren

En del Smalltalkprogrammerare hävdar att den främsta orsaken till att de övergav sin gamla, många gånger avancerade, programmeringsmiljö var Smalltalks mycket kraftfulla och eleganta avlusare. Den ger dig möjlighet att på ett enkelt sätt följa programflödet och även rätta fel och fortsätta exekveringen då felet är rättat utan att behöva starta om applikationen. Det är också enkelt att titta på och förändra variabler som programmet använder.

Som exempelkod använder vi den rekursiva varianten av metoden hailstone: från sidan 400. Vi inför också en metod hailstone som förser metoden hailstone: med ett skönsvärde på argumentet så att det blir enklare att anropa metoden.

```
Integer>>hailstone
    "Anropar metoden hailstone: så att antal steg blir max 20"
    ^self hailstone: 20
```

Om vi startar avlusaren då vi beräknar

```
23 hailstone
```

så kan den se ut som i figur 12.4.

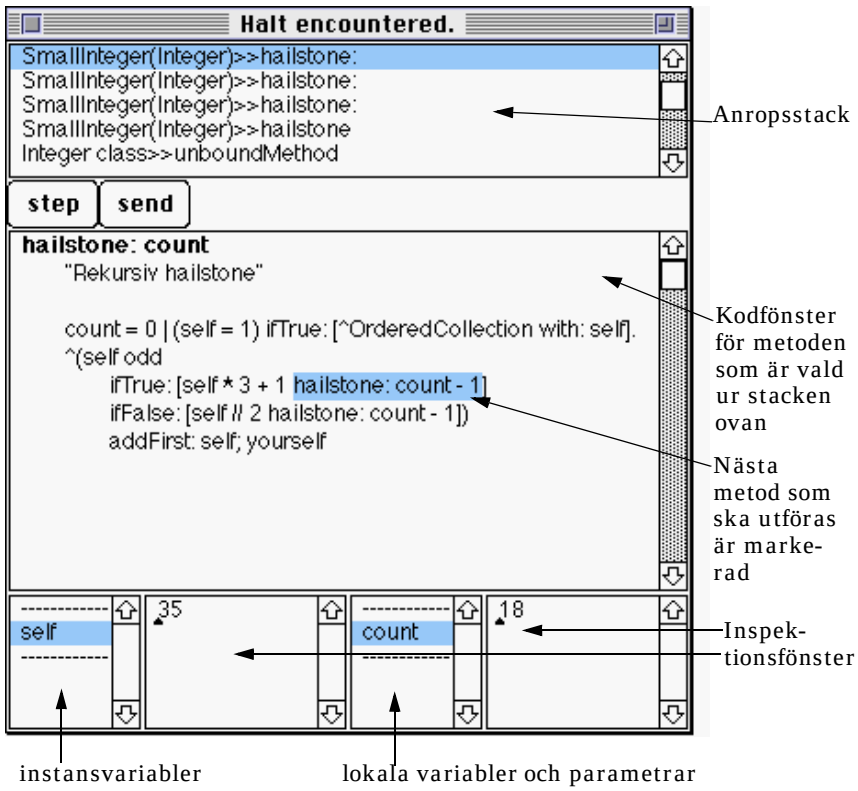
Det översta delfönstret visar anropsstacken. Här ser vi att metoden hailstone: har anropats tre gånger och att det första anropet skedde från metoden hailstone. Texten unboundMethod visar att vi skrev in uttrycket i ett arbetsfönster.

Då en rad i översta delfönstret är markerad så syns dess kod i kodfönstret. Nästa metod som ska utföras eller den metod som nu utförs är markerad.

Den understa delen av avlusaren består av två inspektionsfönster. Det vänstra beskriver self dvs mottagaren av meddelandet som utförs. Det högra beskriver den aktuella metoden med parametrar och lokala variabler.

Anropsstacken

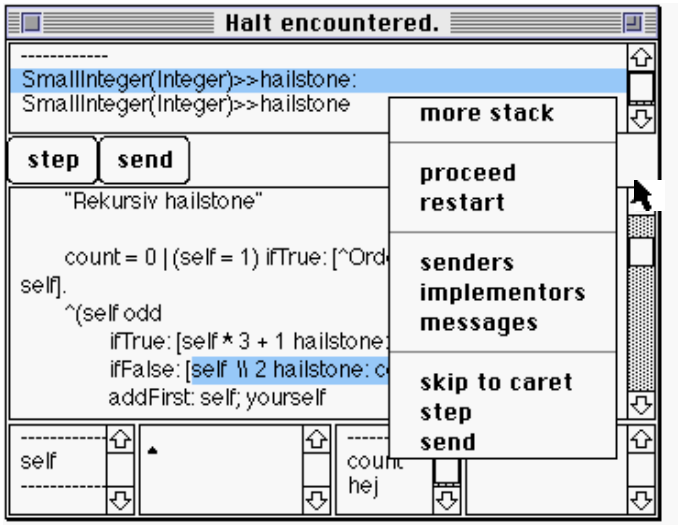
Den översta delen av avlusarfönstret beskriver anropsstacken, med det senaste anropet överst och ”äldre” längre ner. Då en rad markeras visas den aktuella metodens definition och inspektionsfönstren längst ned uppdateras så att de återspeglar det aktuella anropet. Texten på varje rad i stacken anger först mottagarens klass, inom parentes var metoden är definierad och till sist metodens namn. På det sättet får man en bra överblick över anropskedjan.



Figur 12.4 Avlusare som utför metoden hailstone:

I anropsstacken finns en popup-meny för att hantera programexekveringen samt vissa sökoperationer:

- **more stack**
  Visar mer av anropsstacken i listdelen, menyalternativet försvinner om hela stacken redan syns.
- **proceed**
  Fortsätter exekveringen från nuvarande position.
- **restart**
  Startar om exekveringen från början av aktuell metod.
- **senders**
  Ekvivalent med **senders** i Browser och syftar på den valda metoden.
- **implementors**
  Ekvivalent med **implementors** i Browser och syftar på den valda metoden.



Figur 12.5 Menyn för anropsstacken

- **messages**
  Ekvivalent med **messages** i browser.
- **skip to caret**
  Fortsätt exekveringen, stanna vid insättningspunktens position.
- **step**
  Utför en meddelandesändning. Samma som knappen **step.**
- **send**
  Följ med anropet av nästa metod. Samma som knappen **send.**

Kodfönstret

I kodfönstret kan definitionen av den aktuella metoden förändras. Om detta görs så startas metoden om från början. Det går också att utföra smalltalkuttryck här och då refererar self till det objekt som tar emot meddelandet. Då går det att utföra operationer motsvarande dem i ett Workspace, dvs med **do it**, **print it** eller **inspect**, fast nu med variablerna i den aktuella omgivningen redan deklarerade och kanske bundna till värden. Detta medför att mottagaren kan förändras genom att meddelanden skickas till den härifrån. Fönstret ger också direkt tillgång till lokala variabler.

Kodfönstret använder samma operationsmeny som Browserns textfönster. Därmed kan avlusaren användas för att förändra den aktuella metodens definition på precis samma sätt som i Browsern.

Inspektionsfönstren

Precis som i andra inspektionsfönster är det i avlusaren också möjligt att undersöka och ändra det aktuella omgivningens variabler, dvs instansvariabler och temporära variabler. Om förändringar görs så kommer dessa finnas kvar i objektet, de är alltså inte knutna till avlusaren. Inspektionsfönstret för de lokala variablerna och parametrarna fungerar lite anorlunda, eftersom förändringar där är knutna till det aktuella anropet så kommer de endast gälla just nu och för den här exekveringen av metoden. Det går att göra förändringar av både lokala variabler och parametrar, men om en parameter ändras så måste hela metoden startas om från början.

Avlusning med brytpunkter

Brytpunkter i program innebär att programmet stoppas vid speciella punkter. Om en brytpunkt påträffas ges programmeraren möjligheter att antingen fortsätta, avbryta eller öppna avlusaren.

För att införa en brytpunkt i Smalltalk så används metodanropet halt där exekveringen ska stoppas. Metoden halt (som är definierad i Object) avbryter exekveringen tillfälligt och ett avbrottsfönster öppnas. Om vi som exempel prövar den rekursiva varianten för att generera hailstoneserier, hailstone: men med ett litet fel infört. Här har vi infört felet att modulo (\\) används istället för heltalsdivision (//).

```
hailstone: count
    "Rekursiv hailstone med spårutskrift"
    count = 0 | (self =1)ifTrue: [^OrderedCollection with: self].
    ^(self odd
        ifTrue: [self * 3 + 1 hailstone: count - 1]
        ifFalse: [self \\ 2 hailstone: count - 1]) addFirst: self; yourself
```

och utför:

```
23 hailstone
```

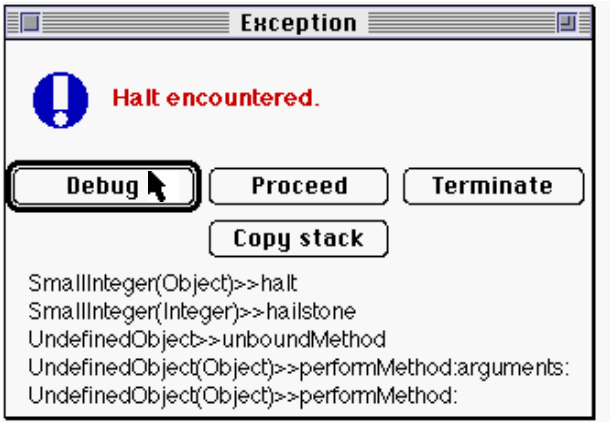
⟹*OrderedCollection (23 70 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0)*

För att avbryta exekveringen och starta avlusaren så kan vi införa en brytpunkt på följande sätt:

```
Integer>>hailstone
    "Anropar metoden hailstone: så att antal steg blir max 20"
    self halt.
    ^self hailstone: 20
```

Om vi på nytt utför 23 hailstone så öppnas avbrottsfönstret i figur 12.6.

Vi kan nu antingen avbryta programmet med knappen **Terminate**, fortsätta direkt med **Proceed** eller starta avlusaren genom att välja **Debug**.



Figur 12.6 Avbrottsfönster som öppnas vid self halt

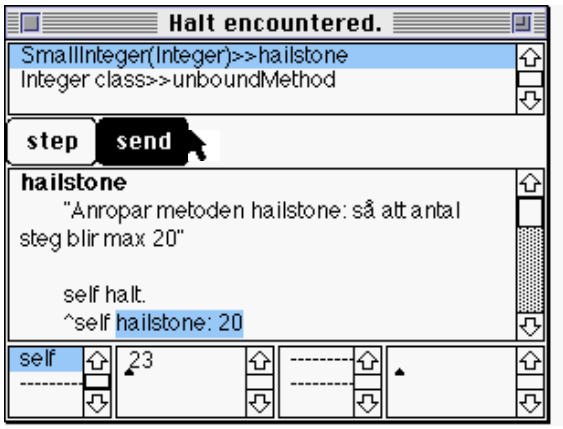
Det finns också ett alternativ **Copy stack** som kopierar anropsstacken, dvs den kedja av meddelandeanrop som gjorts tills dess avbrottsfönstret öppnades. Nederst i avbrottsfönstret syns toppen på anropsstacken.

Vid klick på **Debug** öppnas avlusaren, se figur 12.7. Vi markerar sedan den näst översta raden i avlusarens anropsstack och textfönstret presenterar den aktuella metodens definition. I avlusaren markeras det meddelande som utförs, i det här fallet halt som resulterade i det aktu-

Figur 12.7 Avlusaren då programmet stannar

ella avbrottet.

Vi låter programmet gå vidare ett steg genom att trycka på knappen **step.** Som resultat av detta markeras också meddelandet hailstone: 20



Figur 12.8 Send väljs för att följa anropet till hailstone:

som är nästa metod på tur att utföras. Vi är intresserade av att följa metodens beräkningar och trycker därför på knappen **send**.

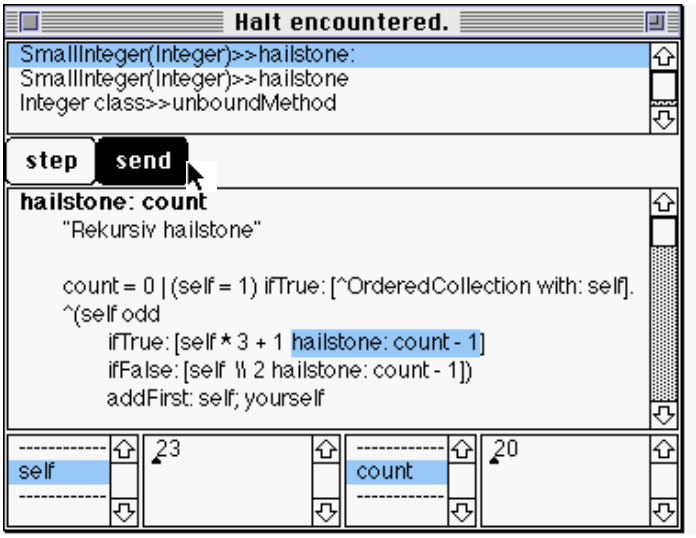
Resultatet av detta är att det markerade meddelandet skickas och avlusarens omgivning byts till den anropade metodens, se figur 12.9. Vi ser i figuren att det mottagande objektet (self) är 23 och argumentet count, som visas i nedre högra delen av fönstret, har värdet 20.

Om vi vill se metodanropet då metoden anropar sig själv, dvs hailstone:, så kan vi göra på två olika sätt. Antingen trycka på knappen **step** tills dess metoden hailstone: är markerad, eller genom att sätta insättningspunkten där vi vill att programmet ska stanna och välja menyalternativet **skip to caret**.

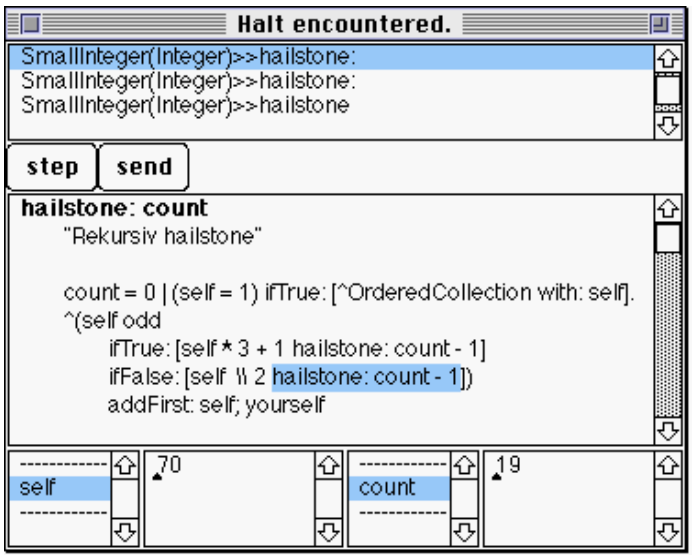
För att följa programmet in i metodanropet så trycker vi på knappen **send**. Nu ser vi det rekursiva anropet till hailstone:. Vi ser nu att self har blivit 70 och argumentet count har minskat med ett till 19, precis som det ska.

Om vi fortsätter med **step** till nästa rekursiva anrop så hamnar vi i delen då self är jämnt. Vi följer det rekursiva anropet med **send**. Denna gång ser vi att self har blivit 0, vilket är fel eftersom vi ville beräkna 100 dividerat med 2 och inte 100 modulo 2.

När man har hittat ett fel i koden med hjälp av avlusaren så kan man naturligtvis öppna en browser, rätta till felet och därefter starta om

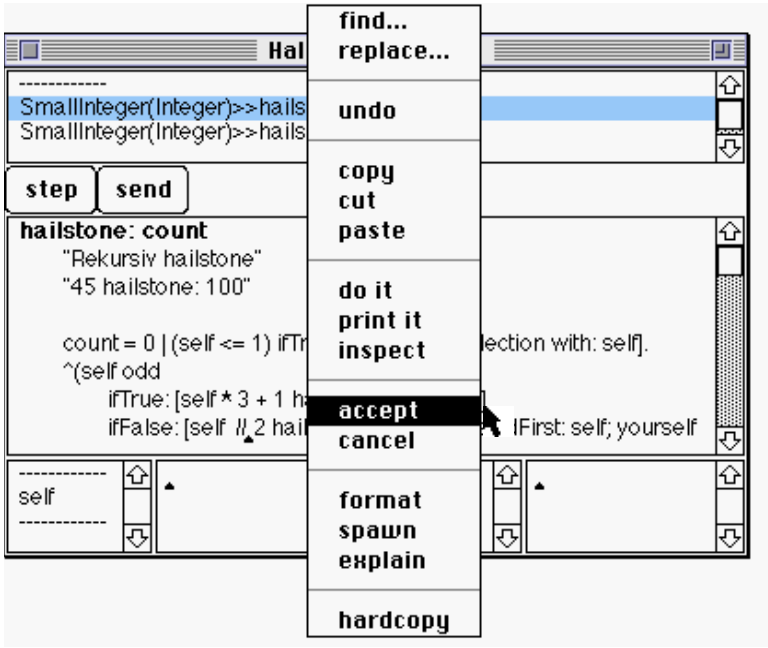


Figur 12.9 Send igen för att följa det rekursiva anropet



Figur 12.10 Rekursivt anrop då self är jämnt

uttrycket som var fel. Men man har också möjligheten att direkt ändra koden i avlusaren och därefter starta om den aktuella metoden.



Figur 12.11 Rätta fel i avlusaren

I avlusaren kan man både hitta fel och rätta dem och därefter fortsätta exekveringen av programmet. Därför innehåller menyn i koddelen av avlusaren samma alternativ som i kodningsdelen av Browsern.

Ett vanligt misstag innan man lärt sig miljön är att man har en browser öppen på en metod som man söker fel i. Samtidigt ändrar man koden direkt i avlusaren på samma metod. Då kommer inte browserfönstret att uppdateras och fortsätter man då att skriva i Browsern och spara koden så kommer alltså ändringen i avlusaren att göras ogjord! Sparar bäst som sparar sist!

12.4 Avbryta exekvering

Det är möjligt att avbryta program genom att trycka på **ctrl-c**. Då kommer ett avbrottsfönster upp som ger möjlighet att starta avlusaren.

Under speciella förhållanden kan det hända att **ctrl-c** inte ”biter” och då kan istället **shift-ctrl-c** användas. Då öppnas ett speciellt fönster där smalltalkuttryck kan evalueras för att rätta till felet. När detta fönster är öppet kan inga andra fönster aktiveras. Kod som skrivs in i detta fönster utförs då **escape-tangenten** trycks in. Då stängs också fönstret.

12.5 Avbrottshantering

Smalltalks avbrottshantering (eng. exception handling) ger oss möjligheter att på ett strukturerat sätt hantera felsituationer. Vi kan ta hand om fel som direkt beror av fel i koden men också om fel som har externa orsaker.

Fördefinierad hantering av avbrott

I många situationer kan det uppstå fel som kanske inte direkt beror av att den egna koden är felaktig utan kanske på att det inte går att komma i kontakt med en viss extern enhet. Ofta önskar man ändå att ett visst kodavsnitt ska utföras, tex för att frigöra referenser till filer. Man vill helt enkelt städa efter sig.

Som exempel ska vi öppna en fil för läsning och därefter läsa ett objekt som är lagrat där mha BOSS och slutligen stänga filen.

```
| readStrm myDict bossStrm |
readStrm := 'MyDictionary.boss' asFilename readStream.
[bossStrm :=BinaryObjectStorage onOld: readStrm.
myDict := bossStrm next]
    valueNowOrOnUnwindDo: [readStrm close]
```

Meddelandet valueNowOrOnUnwindDo: skickas till ett block och ska utföras när blocket är färdigt oavsett hur blocket blev färdigt. Nu vet vi att även om det blir fel i blocket som läser från filen så kommer meddelandet close skickas till readStrm. Meddelandet valueOnUnwindDo: fungerar på ett liknande sätt men argumentblocket utförs då *endast* om ett fel uppstår, inte annars. Se också sidan 199.

Generell hantering av fel

Avbrottshanteringen i Smalltalk baseras på att systemet signalerar ett avbrottsobjekt om något går fel. Avbrottsobjektet letar sig upp i anropsstacken tills dess att en hanterare för det aktuella felet hittas. En sådan felhanterare skapas genom att beskriva vad som ska inträffa om ett viss typ av fel inträffar när ett visst kodavsnitt utförs. Det går tex att specificera vad som ska hända om ett visst meddelande inte förstås, vid fel i kommunikation med externa resurser eller fel pga av att en viss operation inte kan utföras. Ett exempel på det senare är att det enkelt går att beskriva vad som ska ske om ett fel uppstår därför att ett tal har dividerats med noll.

Fel och avbrott hanteras med vanliga Smalltalkobjekt. En typ av objekt är de som signalerar ett fel, instanser av Signal, och en annan typ de som beskriver och hanterar felet, instaner av Exception. Som vi ska se senare är det enkelt att beskriva hur fel ska hanteras om de uppstår. Det är också enkelt att signalera att ett visst fel har inträffat. För det senare går det både att använda redan tidigare definerade felsignaler eller skapa nya.

För exempel på hur fel signaleras rekommenderar vi läsaren att titta i referensmanualen och på meddelandet raise används, dvs utför tex Browser browseAllSendersOf: #raise.

Felhanteringsmall

För att ta hand om fel som inträffar i ett visst kodavsnitt kapslas koden in i en handle:do:-struktur. Schematiskt ser detta ut som följer:

```
felsignal
    handle: [kod som utförs om signalen felsignal inträffar]
    do: [koden]
```

där blocket som ges till nyckelordet do: är den vanliga applikationskoden och koden efter nyckelordet handle: beskriver hur det aktuella felet ska hanteras.

Det går också att ange att bara fel som orsakats av ett visst objekt ska fångas. Då används istället meddelandet handle:from:do:, där den godtagbara felkällan ges som argument till nyckelordet from:.

Hierarki av felsignaler

De olika felsignalerna bildar en hierarki där fel högre upp i hierarkin är mer generella än dem längre ner, se figur 12.12.

```
Object errorSignal
    Object notFoundSignal
        Object indexNotFoundSignal
            Object subscriptOutOfBoundsSignal
            Object nonIntegerIndexSignal
        Dictionary keyNotFoundSignal
        Dictionary valueNotFoundSignal
    Object messageNotUnderstoodSignal
    Object subclassResponsibilitySignal
    ArithmeticValue errorSignal
        AithmeticValue divisionByZeroSignal
        ArithmeticValue domainErrorSignal
    Stream positionOutOfBoundsSignal
    Controller badControllerSignal
    Object informationSignal
        Object notifySignal
        Stream endOfStreamSignal
    Object controlInterruptSignal
        Object userInterruptSignal
    OSErrorHolder errorSignal
    ... (många olika typer av fel från operativsystemet)
```

Figur 12.12 Felsignalstypernas trädstruktur

Figuren visar endast ett utsnitt av de existerande felsignalerna. För fler signaler hänvisar vi till manualen och naturligtvis till smalltalksystemet självt. I systemet kan du tex leta reda på alla ställen där meddelandet newSignal skickas. En ny signal skapas vanligen utifrån en existerande signal och blir då barn till den signalen.

I figur 12.13 beskriver vi de meddelanden som används för att skapa nya signaler utgående från redan tidigare existerande. Om en signal skapas på detta sätt kommer den bli barn till den signal som den skapades ifrån.

newSignal	skapa en ny signal utgående från en existerande (mottagaren av meddelandet)
newSignalMayProceed: proceedBoolean	skapa en ny signal och ange om exekveringen kan fortsätta efter att den signalerats

Figur 12.13 Meddelanden för att skapa nya signaler

Exempel: Ny egen signal

Om vi tex vill skapa en ny signal som barn till Integer divisionByZeroSignal och som inte tillåter att exekveringen fortsätter efter det att den har signalerats kan vi helt enkelt utföra följande:

```
minNyaSignal := Integer divisionByZeroSignal newSignalMayProceed: false
```

Signalera fel

Ett fel signaleras vanligen genom att meddelandet raise skickas till önskad signal, som i följande fall där Object errorSignal signaleras:

```
Object errorSignal raise
```

Det finns också möjligheter att både vid skapandet av signalen och signalerandet av den ange om det ska vara möjligt att fortsätta exekveringen eller om den ovillkorligen ska avbryta exekveringen. Det går också att ge parametrar till signalen med mer eller tydligare information av vad som har inträffat. Det senare kan sedan den felhanterare som tar hand om signalen utnyttja.

raise	signalera mottagaren. Exekveringen kan ej fortsätta efteråt
raiseErrorString: aString	signalera med felmeddelande givet av aString
raiseFrom: parameter	signalera och ange parameter som källa till signalen
raiseRequest	signalera mottagaren. Låt exekveringen fortsätta om felhanteraren önskar
raiseRequestErrorString: aString	som ovan fast med angivande av felmeddelande
raiseRequestFrom: parameter	signalera mottagaren och ange parameter som källa till signalen. Möjligt för felhanteraren att fortsätta exekveringen
raiseRequestWith: parameter	signalera mottagaren och ange parameter som parameter till felet Möjligt för felhanteraren att fortsätta exekveringen

Figur 12.14 De vanligaste meddelandena för att signalera fel

raiseRequestWith: parameter errorString: aString	parameter är parameter och aString felmeddelande. Möjligt för felhanteraren att fortsätta exekveringen
raiseWith: parameter	signalera mottagaren och ange parameter som parameter till felet
raiseWith: parameter errorString: aString	parameter är parameter och aString felmeddelande

Figur 12.14 De vanligaste meddelandena för att signalera fel

Vilka fel fångas?

Som vi nämnde ovan hanteras ett fel med en handle:do:-struktur, där mottagaren är den signal som vi vill fånga. Om vi anger att vi vill fånga en viss typ av fel så kommer vi också fånga de fel som är längre ner i hierarkin för den aktuella signalen.

Exempelvis med följande kod:

```
ArithmeticValue errorSignal
    handle: [hantering av felet]
    do: [koden]
```

kommer vi inte bara hantera fel av typen ArithmeticValue errorSignal utan också de båda felen AithmeticValue divisionByZeroSignal och ArithmeticValue domainErrorSignal.

Vi kan också genom att ange klassen för felsignalen ange att endast fel orsakade av objekt som är instanser av den aktuella klassen eller dess subklasser ska hanteras. Så om vi istället skriver:

```
Integer errorSignal
    handle: [hantering av felet]
    do: [koden]
```

så kommer endast fel som förorsakas av heltal att hanteras. Till skillnad från det föregående exemplet där fel från alla typer av tal eller punkter fångas.

Hanterare av felsignaler kan också kapslas in i varandra, så att vanligen en mer specifik felhanterare befinner sig innuti en mer generell.

Hur hanteras felavbrottet?

När ett fel väl har inträffat ges en beskrivning av det, i form av en instans av Exception, som argument till handle-blocket. Genom att skicka meddelanden till denna instans kan vi ange vad som ska returneras som resultat från koden, ange att det aktuella felet inte ska tas

omhand av det aktuella handle-blocket utan av eventuellt omslutande, starta om do:-blocket, starta ett nytt kodblock och mycket mer.

proceed	fortsätt med nästa sats i den kod som resulterade i felet
proceedDoing: aBlock	argumentblocket anger vad som ska bli resultat efter att handle-blocket har utförst
proceedWith: aValue	resultatet av satsen som resulterade i felavbrottet anges av argumentet
reject	ta inte hand om felet här utan låt omslutande felhanterare göra detta
restart	starta om do:-blocket
restartDo: aBlock	starta om men nu med aBlock
return	avbryt koden och returnera nil
returnWith: aValue	avbryt koden och returnera argumentet som resultat

Figur 12.15 Meddelanden till Exception

Exempel: Fortsätta efter fel

```
Integer errorSignal
    handle: [:exception | exception proceed]
    do: [ 2 / 0.
        'slut']
```

⇒*'slut'*

Exempel: Fortsätta efter fel med angivande av resultat från den felande operationen

```
Integer errorSignal
    handle: [:exception | exception proceedWith: 2]
    do: [ |  x |
        x := 2 / 0.
        x * 4]
```

⇒*8*

Exempel: Felhantering innuti annan felhanterare med släppande av kontrollen

Ett lite mer komplicerat exempel där vi har ett inre handle:do:-block som undersöker vilken metod som gick fel och under vissa villkor släpper felhanteringen vidare och en yttre, mer generell, felhanterare som undersöker i vilken klass felet uppstod.

```
| x |
x := ArithmeticValue errorSignal
    handle: [:exception | exception returnWith: exception originator]
    do: [Integer errorSignal
            handle: [:exception | exception parameter selector = #/
                        ifTrue: [exception reject]
                        ifFalse: [exception returnWith: 0]]
            do: [ 2 / 0]].
x
```

⟹*Fraction*

Exempel: Division med noll version 1

Som exempel på hur man kan hantera fel så skriver vi ett kodavsnitt (här i ett arbetsfönster) där vi tar om hand om division med noll och skriver ut en felutskrift i utskriftsfönstret om det inträffar.

```
| a b |
a := 0.
b := 0.
ArithmeticValue divisionByZeroSignal
    handle:
        [:theException |
        Transcript show: 'Kvoten kunde inte beräknas; division med noll'; cr.
        theException return]
    do: [Transcript show: 'kvoten mellan ', a printString , ' och  ' , b printString,
' är ' , ( a / b) asFloat printString ; cr].
```

⟹'Kvoten kunde inte beräknas; division med noll

Med den här lösningen kommer alltså ett eventuellt felavbrott vid division med noll ersättas med ett felmeddelande i utskriftsfönstret.

Exempel: Division med noll version 2

För att ge användaren av blocket en möjlighet att korrigera en nämnare som är noll skriver vi om exemplet på förljande sätt:.

```
| a b |
a := 3.
b := 0.
ArithmeticValue divisionByZeroSignal
    handle:
        [:theException |
```



```
            b := (Dialog
                    request: 'Nämnaren var noll, ge nytt värde på nämnaren')
                                                                asNumber.
            theException restart]
        do: [
    Transcript show: 'kvoten mellan ', a printString , ' och  ' , b printString , '  är ' , ( a
    / b) asFloat printString ; cr].
```

⇒Dialog där nytt värde för nämnaren ges; 2.

⇒kvoten mellan 3 och 2 är 1.5

Exempel: Division med noll version 3

Man skulle också kunna tänka sig en lösning där man vill returnera kvoten om den går att beräkna annars vill man returnera 0 om både täljare och nämnare är 0 och om bara nämnaren är 0 så vill man att felsignalen ska fortsätta.

```
| a b kvot |
a := 0.
b := 0.
kvot := ArithmeticValue divisionByZeroSignal
            handle:
                [:theException |
                a = 0 ifTrue: [theException returnWith: 0].
                theException reject]
            do: [a / b]
```

Om vi nu utför koden för olika värden på a och b så blir resultatet:

a	**b**	**resultat**
3	2	1.5
0	0	0
1	0	Felavbrott

12.6 Projekt

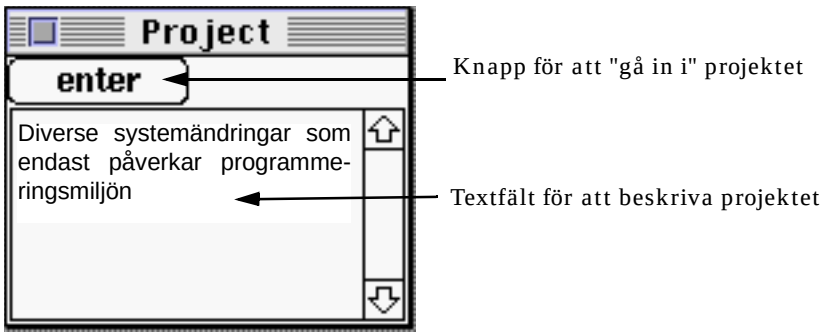
Det är ofta svårt att i stora system veta precis vilka tillägg som har gjorts för en viss applikation. Om vi till exempel har gjort några tillägg i klasserna Object och Integer så måste vi tänka på att spara även dessa tillägg om vi vill spara koden för den aktuella applikationen. För att underlätta för programmeraren så finns det ett delsystem i Smalltalk som håller reda på alla förändringar som gjorts i systemet. Objekten som hanterar förändringsmängder är instanser av klassen ChangeSet.

Ofta är det bra att placera olika klasser eller olika delar av en applikation i olika förändringsmängder. Detta kan enkelt göras genom att pro-

ject används. Till varje projekt är förutom ett ChangeSet även en ny uppsättning fönster kopplade. Det betyder att då man går in i ett projekt så kommer alla gamla fönster temporärt stängas och alla fönster kopplade till det nyöppnade projektet att öppnas. Alla förändringar som görs i ett projekt kommer bara sparas i detta projekts förändringsmängd och inte i övriga förändringsmängder. Det är endast fönsteruppsättningen och förändringsmängden som skiljer mellan de olika projekten inte något annat. Källkod, globala variabler mm delas alltså mellan projekten.

För att hantera ett ChangeSet kan menyn **Changes** i Smalltalks huvudfönster användas. Alternativen är:

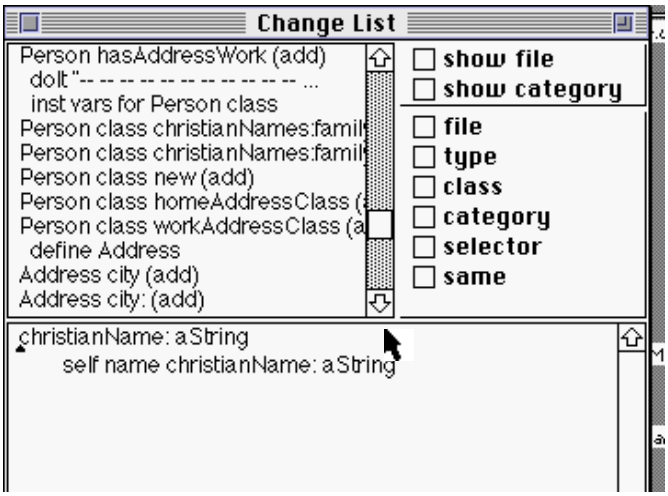
- **Inspect ChangeSet**
  Ett inspektionsfönster öppnas på förändringsmängden.
- **Empty Changes...**
  Tömmer förändringsmängden så att den inte längre kommer ihåg de förändringar som gjorts tidigare.
- **File Out Changes...**
  Skriver aktuellt ChangeSet på fil.
- **Open Project**
  Skapa ett nytt ”barnprojekt” till det aktiva projektet. Det kommer då öppnas ett fönster med en ”ENTER” knapp i som man trycker på för att gå in i projektet, se figur 12.16.
- **Exit Project**
  Lämna det nuvarande projektet och gå till föräldraprojektet.



Figur 12.16 Projektfönster

12.7 Förändringslista

En förändringslista (eng: changelist) är ett hjälpmedel för på ett kontrollerbart sätt läsa in smalltalkkod från fil . Om man öppnar en förändringslista från en fil med smalltalkprogram i så kommer koden att analyseras och varje klassdefinition, metoddefinition eller smalltalkuttryck blir en egen rad i listan.



Figur 12.17 ChangeList från Person-exemplet på sidan 362

För att läsa in kod till listan eller skriva ut från den, till fil så använder man menyoperationerna längst upp i listdelens popup-meny. Dessa är

- **read file/directory...**
  Läser in kod från den fil eller från alla filer i den katalog specificeras
- **write file...**
  Skriver ut den kod som syns i listan på en fil
- **recover last changes**
  Läser in alla ändringar som gjorts sedan imagen sparades
- **display system changes**
  Bygger upp en lista med alla ändringar som finns registrerade i aktuellt ChangeSet

Vi ser i figur 12.17 alla klassdefinitioner, metoddefinitioner och initieringar i en lång lista. För att kontrollera vad som ska presenteras kan filter anges i de olika kryssrutorna. För att välja ut tex all kod i en viss klass så markerar man en rad ur klassen (metod- eller klassdefinition) och därefter kryssar man i **class.** Nu kommer bara metoder och defini-

tioner av klassen finnas i listan. När man har valt ut en del av listan kan man med mittknappsmenyn välja olika operationer på listan. Man kan dela upp dessa i två grupper, en som utför något för alla rader i listan och den andra som bara utförs på den markerade raden.

Om vi tex vill läsa in allt *utom* en viss klass från en fil i smalltalksystemet så öppnar vi först en ChangeList på den filen. Därefter markerar vi en metod i den klass som man inte vill ha med och sedan kryssar vi för **class**. Nu kommer bara den klassen vi inte vill läsa in synas, vi utför menyalternativet **remove all** och alla klassens rader borttagsmarkeras. Nu kan vi avmarkera kryssrutan **class** och åter se all kod från filen.

Att en klass och dess metaklass ej är densamma avspeglar sig i hanteringen av förändringslistan. Så filter eller operationer som appliceras på en viss klass gäller inte dess metaklass och vice versa. Så om vi vill ta bort en viss klass inklusive dess metaklass kan vi gå tillväga på följande sätt:

1 markera någon metod i klassen
2 välj klassfiltret (**class**)
3 ta bort alla metoder (**remove all**)
4 ta bort klassfiltret
5 välj någon metod i metaklassen
6 välj klassfiltret igen
7 ta bort alla metoder.

Ett tips för att hitta eventuella klassmetoder är att de oftast är definierade direkt efter instansmetoderna i filen. Nu är det dags att läsa in resten av filen och det gör man med menykommandot **replay all**.

Menyn som hanterar operationer på listan är:

- **replay all**
  Läser in alla rader i listan
- **remove all**
  borttagsmarkerar allt i listan, dvs dessa kommer inte utföras om replay utförs
- **restore all**
  tar bort borttagsmarkeringen för alla borttagsmarkerade alternativ
- **forget**
  Tar bort alla rader som är borttagningsmarkerade från listan så att de inte längre finns med
- **replay selection**
  Läser in raden som är markerad
- **remove selection**
  Borttagsmarkerar den markerade raden

- **restore selection**
  tar bort borttagsmarkeringen av den markerade raden

Vi har redan använt kryssrutan **class** vid sidan av listan, men det finns också andra kryssrutor. Om man markerar flera samtidigt så ger det ytterligare restriktioner på vilka rader i listan som ska synas. Kryssrutorna är:

- **file**, alla rader som kommer från samma fil. Man kan alltså läsa in flera filer efter varandra i en ChangeList
- **type**, alla av en viss typ som, metoddefinition, klassdefinition, smalltalkuttryck eller kommentar
- **class**, alla ur en viss klass
- **category**, alla som är definierade i samma kategori. Markeras en klassdefinition får klasskategori, markeras en metod får man metodkategorin.
- **selector**, alla metoder som har samma namn.
- **same**, alla som är lika. Om man väljer en metod betyder det alla eventuella omdefinitioner av metoden i listan

Vi inläsning av kod kan det bli konflikter mellan det existerande systemet och den nya koden. För att kontrollera det har man också ta hjälp av ChangeList.

Detta kan göras på två olika sätt:

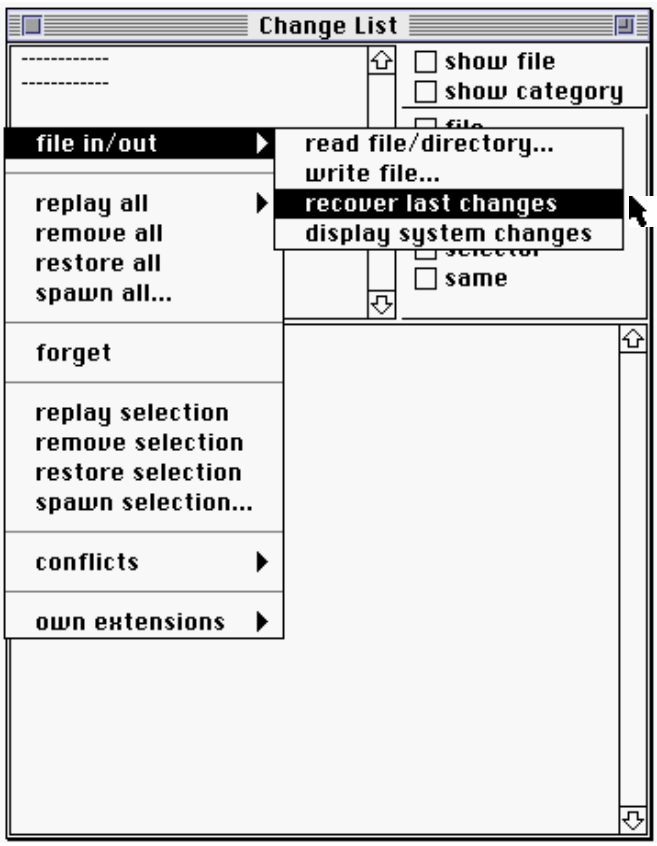
1 titta på varje rad i listan och se om det sista ordet på raden, som står inom parentes, är changed eller add. Det säger om raden förändrar något i systemet eller om den lägger till kod.
2 det andra sättet är att i list-delens meny välja alternativet **conflicts->check conflicts with system, save as...**.
  Om detta alternativ väljs skapas en fil som beskriver konflikterna mellan koden och det aktuella systemet.

12.8 Återstart efter krasch

En vanlig situation då ChangeList används är för att återställa systemet efter en eventuell systemkrasch.

För att återskapa systemet sedan det senast sparades så öppnas en ChangeListView och i list-delen väljs sedan menyalternativet **file in/out->recover last changes.**

Från loggfilen skapas då en lista på allt som gjorts sedan systemet senast sparades. Listan innehåller metoddefinitioner klassdefinitioner men också Smalltalkuttryck som har utförts. Förmodligen vill man inte



Figur 12.18 Återstart efter krash

utföra Smalltalkuttrycken och borttagsmarkerar dem på vanligt sätt. Nu kommer bara koddefinitioner att vara kvar. Dessa kan nu läsas in med menyalternativet **replay all**.

En krasch kan ha orsakats av en felaktigt definierad metod eller klass (som kanske är resultatet av experimenterande med systemet). Det är naturligtvis viktigt att en sådan definition inte återigen läses in i systemet. Eftersom listan skapas utifrån changes-filen så kommer listan vara kronologisk ordnad med de senaste definitionerna i slutet av listan. Det betyder också att källan till kraschen ofta finns i slutet av listan.

Sammanfattning

Avlusning

Spårutskrifter programsatser med utmatning i utskriftsfönstret för att följa exekveringen.

Inspektionsfönster speciellt fönster för att studera detaljer hos enskilda objekt.

Avlusaren för att stegvis följa exekveringen. Om du ändrar kod direkt i avlusaren, kom då ihåg att andra fönster mot samma metod inte uppdateras.

Brytpunkter med self halt för att starta avlusaren.

Programmeringsstil

Avbrottshantering: valueNowOrUnwindDo: för att alltid utföra argumentblocket, även om något blir fel. Meddelandet valueOnUnwindDo: om argumentblocket endast skall utföras om ett fel inträffar.

Fånga felsignaler om inte andra sätt är tillämpbara eller det är väldigt viktigt att det inte går fel.

Miljön

Projekt för att hålla isär olika tillägg man gör till systemet.

ChangeList för att hålla reda på förändringar, läsa in kod och återskapa system efter krascher.

Smalltalk

Alla *objekt kan inspekteras* genom att skicka meddelandet inspect eller basicInspect till dem. Det första resulterar i en eventuellt klassspecifikt inspektionsfönster.

`Ctrl-C` kan användas för att avbryta exekveringen.

`Shift-Ctrl-C` bryter med högre prioritet än **`Ctrl-C`**.

Signal används för att skapa felsignal, dess instanser signaleras med olika varianter av meddelandet raise.

Exception beskriver hur felsignaler hanteras.

För att *fånga fel* som genereras av signaler används följande mall:

```
felsignal
    handle: [:exception | kod som utförs om signalen felsignal inträffar i
                          do:-blocket]
    do: [koden]
```

Övningar

12.1 Prova med att avsiktligt avbryta arbetet i din image och därefter återskapa systemet med hjälp av changes-filen genom att

a) Gå ur smalltalksystemet utan att spara.

b) Starta om datorn utan att spara.

13 Processhantering

Detta kapitel besvarar bland annat följande frågor:

- Vad är parallellitet?
- Vad innebär processer i Smalltalk?
- Vad är en lättviktsprocess?
- Vad är en semafor?
- Hur kan man tillfälligt stoppa ett program en viss tid?

I det här kapitlet ska vi beskriva hur vi kan konstruera och kontrollera vad som brukar kallas lättviktsprocesser. Det betyder att det går att låta olika programdelar köra oberoende av varandra, på samma sätt som datorers operativsystem tillåter flera program att köra samtidigt. Det finns även implementationer av Smalltalk som kör på flera processorer utan att programmeraren behöver ta särskild hänsyn till det.

13.1 Processer

För att hantera processer finns det två centrala klasser, Process och ProcessScheduler. Objekt ur klassen Process beskriver ett program som kör, med bla dess programpekare och lokala variabler. Klassen ProcessScheduler har en instans i form av den globala variabeln Processor. Detta objekt hanterar alla olika processer i systemet inklusive vilken process som ska få köra vid ett visst tillfälle.

De processer som skapas är sk lättviktsprocesser. Det betyder att processerna befinner sig i samma adressrymd, de kan komma åt samma objekt och är därför inte skyddade från varandra. Det betyder också att de enkelt kan komunicera genom att dela en variabel. Som jämförelse kan nämnas att processer under Unix normalt inte är av lättvikstyp och därmed delar de inga data och speciella operativssystemkommandon måste användas för att föra information mellan dem.

Processkontroll

En lättviktsprocess skapas från ett block. Blocket uppmanas att bli en process genom att meddelandet fork skickas till det.

Exempel: klocka

Som exempel skapar vi en klocka som skriver ut tiden i utskriftsfönstret en gång per minut. Klockan startar vi genom att instansiera klassen Clock och därefter skicka meddelandet start till det nya objektet. Vi kommer i avsnitt 13.2 beskriva hur Clock kan implementeras

```
| clockProcess clock|
clockProcess :=
    [clock := Clock new.
    clock start] fork.
Transcript show: 'En ny tidprocess har startat' ; cr
```

Den nya processen skapas och startas i ett enda steg då blocket får meddelandet fork. Variabeln clockProcess kommer peka på den nya processen. Efter att processen skapats så skrivs meddelandet ”En ny tidprocess har startat” ut i utskriftsfönstret. Eftersom Clock-processen körs oberoende av processen som skapade den så kan man inte säkert veta tidsordningen mellan utskriften 'En ny tidprocess har startat' och den första tidsangivelsen från den nya processen. Ordningen är alltså obestämd.

Det är möjligt att påverka processer genom att skicka meddelanden till dem. Det går tex att stoppa, återstarta och döda processer. En process stoppas genom att meddelandet suspend skickas till den. Resultatet är att processen temporärt stoppas. För att återstarta en process så skickas meddelandet resume till den. För att stoppa en process permanent så att den inte åter går att starta så skickas meddelandet terminate till den.

Om vi vill stoppa klockan så utför vi alltså.

```
clockProcess suspend
```

Klockan kommer nu vila tills dess att vi åter uppmanar processen att starta

```
clockProcess resume
```

Det är också möjligt att skapa processer som inte omedelbart startas. Det kan göras gemom att meddelande newProcess skickas till ett block. Klockexemplet blir då

```
clockProcess :=
    [clock := Clock new.
    clock start] newProcess.
```



```
clockProcess resume.
```

Processprioritering

Smalltalk använder en enkel form för att bestämma vilken process som ska få köra. Grunden är att en process får köra tills dess att den frivilligt lämnar ifrån sig kontrollen. När det är dags för en ny process så är det den process som har fått vänta längst som får fortsätta. Den här principen för schemaläggning av processer brukar på engelska kallas *non preemtive round robin*. För att tala om att någon annan process får köra så skickas meddelandet yield till Processor.

Förutom denna enkla princip så styrs fördelningen av processortid också av prioritetsnivåer. Här gäller att en process med högre prioritet har företräde även om en lägre prioriterad process inte har avsagt sig kontrollen.

Eftersom en högre prioriterad process alltid har företräde är det viktigt att inte ge processer för hög prioritet eftersom det kan blockera hela smalltalksystemet. I systemet finns 100 olika prioritetsnivåer men det är bara ett fåtal av dessa som används. De som används har symboliska namn definierade i klassen ProcessScheduler och är beskrivna i figur 13.1:

timingPriority	100
highIOPriority	98
lowIOPriority	90
userInteruptPriority	70
userSchedulingPriority	50
userBackgroundPriority	30
systemBackgroundPriority	10
systemRockBottomPriority	1

Figur 13.1 Processernas prioritetsnivåer

När en ny process skapas tex via meddelandet fork kommer den att få samma prioritet som processen som skapade den. För att skapa en process med en annan prioritet så kan metoden forkAt: användas.

Exempel: klockprocess med angiven prioritet

Starta klockprocessen på högre nivå än standardnivån userSchedulingPriority

```
clockProcess := [Clock start] forkAt: Processor userInteruptPriority.
```

Med den här definitionen kommer klockan skriva ut tiden även om en normal system- eller användarprocess är mitt inne i en tidsödande beräkning. Fördelen med detta är, att tiden kommer att uppdateras även om andra ”normala” processer exekverar.

Metoder för att skapa och hantera lättviktsprocesser

Skapa lättviktprocess

En lättviktsprocess skapas genom ett av fyra olika meddelanden skickas till ett block som i sin tur beskriver processens beteende.

Vi sammanfattar de viktigaste meddelandena i figur 13.2.

fork	Skapar och startar ny process
forkAt: prioritet	Skapar och startar ny process med angiven prioritetsnivå
newProcess	Skapar ny process
newProcessWithArguments:	Skapar ny process och för in argument motsvarande valueWithArguments:

Figur 13.2 Metoder i BlockClosure som hanterar processer

Kontrollera lättviktsprocess

Det går att starta, stoppa och döda processer. En process har också en prioritetsnivå som både kan avläsas och ändras. En stoppad process kan startas igen från de läge den stoppades i.

resume	startar process
suspend	stannar process
terminate	dödar process
priority	returnerar prioritetsnivån
priority:	sätter prioritetsnivå

Figur 13.3 Metoder i Process

Prioritetsnivåer för lättviktsprocesser

Processer hanteras av den globala Processor som är instans av ProcessorScheduler. Här definieras bland annat prioritetsnivåer och schemaläggningen av dem sker här.

yield	ger kontrollen till andra processer med samma prioritet som den aktiva processen
activeProcess	returnerar aktiv process
activePriority	processens prioritet
terminateActive	dödar aktiv process
timingPriority	högsta prioritetsnivå. Används av systemprocess som hanterar realtid
highIOPriority	för tex hantering av nätverk
lowIOPriority	hantering av användarinmatning, mus, tangentbord etc
userInteruptPriority	processer som vill ha omedelbar service
userSchedulingPriority	standard prioritetsnivå. Används bla av fönsterhanteraren
userBackgroundPriority	användarens bakgrundsprocesser
systemBackgroundPriority	systemets bagrundsprocesser som optimerande kompilator
systemRockBottomPriority	lägsta möjliga prioritetsnivå

Figur 13.4 Metoder i ProcessorScheduler

13.2 Fördröjning

För att temporärt stoppa en process så kan en instans av Delay skapas. Därefter skickas meddelandet wait till den.

```
| fördröjare |
fördröjare := Delay forSeconds: 55.
fördröjare wait.
```

Om koden utförs, så stoppas den upp i minst 55 sekunder.

När ett objekt ur klassen Delay skapas så anges hur länge processen ska vila. Antingen i hela sekunder eller millisekunder.

Med hjälp av Delay kan vi nu skapa en Clock-klass som skriver ut tiden i utskriftsfönstret.

```
Object subclass: #Clock
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Clocks'
```

KLASSMETODER

```
start
    "Skriver ut tiden i Transcript en gång per minut"
    [true]
        whileTrue:
            [Transcript show: Time now printString; cr.
            (Delay forSeconds: 60 - Time now seconds) wait]
```

På sista raden skapas en fördröjning tills nästa gång det sker en minutövergång. Direkt efteråt så skickas meddelandet wait så att processen kommer i vila. Det går också bra att använda meddelandet forMilliseconds:, för att skapa fördröjningen, om större noggrannhet önskas.

Under tiden som processen vilar kan andra processer med samma eller lägre prioritet köra.

Metoder för att skapa fördröjningar

Konstruera fördröjning

Det finns två olika sätt att skapa instanser av Delay.

forMilliseconds: tid	skapa en fördröjning angiven i millisekunder
forSeconds: tid	skapa fördröjning angiven i hela sekunder

Figur 13.5 Klassmetoder i Delay

Utför fördröjning

Metoden wait är den metod som normalt används för att utföra en fördröjning.

wait	stanna den tid som angavs då processen skapades
inProgress	väntar processen fortfarande?

Figur 13.6 Instansmetoder i Delay

13.3 Semaforer

Klassen Semaphore ger oss möjligheter att synkronisera processer med varandra. Med hjälp av semaforer så kan en process fås att vänta på en annan process. Det finns två primära metoder som semaforer förstår: signal och wait. Internt i en semafor finns en räknare som räknas upp då meddelandet signal skickas till den och ner om wait skickas till den. Vidare gäller att om räknaren är noll och någon skickar wait så kommer processen stoppas tills dess någon annan process skickar meddelandet signal till den.

För att exemplifiera semaforer utgår vi ifrån Clock-exemplet. Vi utökar klassen Clock så att den förutom att skriva ut tiden också tillfälligt kan skriva ut dagens datum genom att meddelandet showDate skickas till den. Datumet kommer då att skrivas ut en gång. När en ny klocka skapas kommer inte datumet att visas. Metoden för att skriva ut tid och eventuellt datum blir nu:

```
showDate
    "Skriver ut tiden i Transcript en gång per minut, första gången med datum"
    [true]
        whileTrue:
            [displayDate
                ifTrue:
                    [Transcript show: Date today printString , ".
                    displayDate := false].
            Transcript show: Time now printString; cr.
            (Delay forSeconds: 60 - Time now seconds) wait]
```

Om vi vill skapa en klocka som visar datumet vid första utskriften så kan en första ansats bli

```
| clockProcess clock|
clockProcess :=
    [clock := Clock new.
    clock start] fork.
clock showDate
```

Om vi utförde denna kod skulle det förmodligen resultera i ett felavbrott av typen "nil does not understand: #showDate". Skälet är att variabeln clock ännu inte har tilldelats den nya klockan inne i blocket. Man måste alltså se till att blocket har startat och tilldelat variabeln clock den nya klockan.

```
| clockProcess clock sem|
sem := Semaphore new.
clockProcess :=
    [clock := Clock new.
    sem signal.
    clock start] fork.
sem wait.
clock showDate.
```

Först skapas en ny semafor, sem. Därefter skapas den nya processen. Här har vi också försäkrat oss om att meddelandet showDate inte utförs förrän variabeln clock har tilldelats den nya klockan. Därmed kommer det inte att bli något felavbrott. Men vad vi inte vet är om utskriften av dagens datum kommer att ske första gången tiden skrivs ut eller senare. För att kontrollera detta ser vi till att clockProcess startas först när clock showDate har utförts.

```
| clockProcess clock sem|
sem := Semaphore new.
clockProcess :=
    [clock := Clock new.
    sem signal.
    sem wait.
    clock start] fork.
sem wait.
clock showDate.
sem signal
```

Naturligtvis går det att skriva om koden så att semaforer inte behövs genom att konstruera klockan innan den processen skapas.

Kritiska sektioner

Klassen Semaphore implementerar metoder som ser till att endast en process åt gången utför ett visst kodavsnitt, sk kritiska sektioner. För att åstadkomma detta skapas en semafor med hjälp av meddelandet forMutualExclusion. Denna semafor kan sedan enkelt användas för att se till att endast en process åt gången exekverar ett visst kodavsnitt. Semaforer förstår meddelandet critical: ettBlock, där ettBlock innehåller den kritiska koden.

Exempel: pipeline

Vi ska titta på ett exempel där semaforer och processkontroll behövs för att föra objekt mellan två eller flera processer på ett flexibelt sätt. Vi konstruerar en kö där en process lägger in objekt och en annan process tar ur objektet. Om det inte finns några objekt i kön så väntar processen tills dess det kommer objekt. Lösningen baseras på en semafor som håller reda på antalet objekt i kön. Semaforen räknar upp då nya element

läggs in och ner då element tas bort. Vidare behöver kontroller göras så att inte två processer samtidigt lägger till eller tar bort ur kön då detta kan leda till att kön blir felaktig. Detta löser vi med kritiska sektioner där endast en process åt gången får köra. Vi kallar klassen för Pipe och definierar den med tre instansvariabler.

```
Object subclass: #Pipe
    instanceVariableNames: 'list criticalSem pipeSizeSem '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Pipe-line'
```

Instansvariabeln list är listan med objekt; criticalSem är semaforen som används för kritiska sektioner; pipeSizeSem stoppar processer som försöker plocka bort från en tom Pipe.

Initiering av ny Pipe görs som vanligt med initialize.

INSTANSMETODER

```
initialize
    list := OrderedCollection new: 10.
    criticalSem := Semaphore forMutualExclusion.
    pipeSizeSem := Semaphore new
```

Vi definierar metoden nextPut:, som ska användas av klienter som vill lägga in nya objekt i listan. Metoden har en kritisk sektion nämligen då ett objekt läggs till instansvariabeln list. Se speciellt hur vi använder meddelandet critical: för att garantera att ingen annan process lägger till objekt samtidigt.

```
nextPut: anObject
    "lägger till ett objekt i listan på ett säkert sätt. Signalerar
    därefter att objektet finns där"
    criticalSem critical: [list addLast: anObject].
    pipeSizeSem signal.
    ^anObject
```

Vi definierar också metoden next, som används för att ta bort objekt från listan. Här börjar vi med att skicka meddelandet wait till pipeSizeSem så att processen stannar om instansen av Pipe är tom. Först därefter, då det finns objekt i listan, tar vi bort och returnerar det första objektet.

```
next
    "väntar först om det inte finns något objekt i listan. Tar bort
    första objektet i listan på ett säkert sätt."
    pipeSizeSem wait.
    ^criticalSem critical: [list removeFirst]
```

Vi skriver också en del andra användbara metoder, som får Pipe att likna en konventionell ström.

```
peek
    "väntar först om det inte finns något objekt i listan. Tar bort
    första objektet i listan på ett säkert sätt."
    ^criticalSem critical: [list isEmpty
        ifTrue: [nil]
        ifFalse: [list first]]

size
    "returnerar antal objekt i pipen"
    ^criticalSem critical: [list size]

isEmpty
    "är pipen tom eller ej"
    ^self size == 0

KLASSMETODER

new
    ^super new initialize
```

Viktiga metoder

Vi sammanfattar det viktigaste metoderna i Semaphore.

Konstruktörer

new	skapar en vanlig semafor
forMutualExclusion	skapar en semafor för kritiska block. Skapar semaforen och signalerar den en gång

Figur 13.7 Konstruktörer för Semaphore

Klassmetoden forMutualExclusion är implementerad på följande sätt:

```
forMutualExclusion
    ^self new signal
```

Som synes är det en helt vanlig semafor som direkt signaleras. En semafor som används för kritiska sektioner bör alltid börja med att signaleras en gång, detta för att den första processen som vill utföra dess kod inte ska stoppas, och därmed alla andra processer också. Då kritiska sektioner är så pass vanliga erbjuds denna konstruktör.

Instansmetoder

signal	signalera till semaforen
wait	vänta tills dess att någon signalerar
critical: ettBlock	endast en process åt gången kan utföra ettBlock

Figur 13.8 Instansmetoder för Semaphore

13.4 Delad kö

I föregående avsnitt definierads klassen Pipe för att föra information mellan processer på ett kontrollerat sätt. Klassen Pipe är i allt väsentligt identiskt med systemklassen SharedQueue. Klassen används för att föra information mellan processer på ett kontrollerat sätt. Den kan i mycket liknas vid Unix pipe, men med den skilnaden att SharedQueue i Smalltalk inte har en max-storlek.

next	ta bort och returnera nästa objekt i listan
peek	nästa objekt i listan
nextPut: ettObjekt	lägger till ettObjekt sist i listan
size	antal element i listan
isEmpty	är listan tom?

Figur 13.9 Användbara metoder i SharedQueue

13.5 Utskriftsexempel

Eftersom processerna i klockexemplet i avsnitt 13.1 kördes oberoende av varandra så blev det problem med att avgöra om utskriften från klockprocessen eller utskriften som talade om att en ny process hade startats skrevs ut först. Det finns också ett annat allvarligare problem med dessa utskrifter i utskriftsfönstret. Problemet är att Transcript's metoder inte hanterar parallella processer. Därmed kan det bli fel om två samtidiga utskrifter görs i utskriftsfönstret. Vi ska titta på två olika metoder att lösa det här problemet: en med delad kö och en som använder kritiska sektioner.

Delad kö och processnivåer

Nu ska vi titta på hur en delad kö kan användas för att få utskrifter i utskriftsfönstret att fungera även om flera processer används. Vi ska-

par en delad kö som sköter kommunikationen med Transcript. Klienterna gör sedan sin utmatning till kön istället för direkt till Transcript.

```
Object subclass: #SharedTranscript
    instanceVariableNames: 'queue process '
    classVariableNames: 'Instance '
    poolDictionaries: ''
    category: 'Shared-Transcript'
```

Instanvariabeln queue som innehåller det som ska skrivas ut och process är processen som flyttar objekt från kön till Transcript. Dessutom finns en klassvariabel Instance som ska innehålla den enda instansen ur denna klass. Om det finns mer än en instans så har vi inte vunnit något eftersom det då finns risk för samtidig utskrift i utskriftsfönstret.

Inspektorer:

```
process
    ^process

queue
    ^queue
```

Meotoden run används för att skapa en process som flyttar strängar från den delade kön till Transcript-fönstret. Den ska anropas en gång då objektet skapas.

```
run
    "Startar en ny process som läser ifrån kön och skriver ut
    innehållet i Transcript-fönstret"
    process := [[self process == Processor activeProcess]
            whileTrue: [Transcript show: self queue next]] newProcess.
    self process priority: Processor userInterruptPriority - 1.
    self process resume
```

Processen som skapas är en while-slinga med ett avbrottsvilkor som anger när processen ska terminera och en kropp som utför den egentliga uppgiften. Avbrottsvilkoret ser till att processen termineras om instansvariabeln process ändras. På detta sätt kan vi som vi ska se i metoden release stoppa processen genom att tex sätta process till nil. Ett alternativ vore att terminera processen genom att skicka meddelandet terminate till den. Men då kan vi om vi har otur bryta mitt inne i en utmatning till utskriftsfönstret.

Vi har i metoden valt att ge utskriftsprocessen en relativt hög prioritet. Skälet till det är att vi vill att utskriften i Transcript ska ske snabbt efter det att man har begärt en utskrift av SharedTranscript. Vi börjar med att skapa en process utan att starta den genom att skicka meddelandet newProcess till blocket, den nya processen tilldelas instansvariabeln process. Därefter ökar vi processens prioritet till ett under avbrottshanterarens nivå. Därigenom är det möjligt att avbryta processen med ctrl-c.

Därnäst startar vi processen. Skälet att vi inte använder metoden forkAt: enPrioritetsnivå, som är det enklaste, är att då skulle den egna processen bli avbruten direkt efter att processen skapades eftersom den nya processen har högre prioritet. Det betyder att metoden inte hinner utföra tilldelningen av instansvariablen process och det leder i sin tur till att den nya processens whileTrue: villkor, process == Processor activeProcess, inte är uppfyllt, och processen terminerar omedelbart.

För att man ska kunna skriva ut text på samma sätt som med vanliga Transcript så implementerar vi nedanstående metoder. Det är långt ifrån alla i Transcript, men det är enkelt att definiera övriga också.

```
cr
    "skriv ut ny rad"
    self queue nextPut: (String with: Character cr)

show: aString
    "skriv ut en sträng"
    self queue nextPut: aString

tab
    "skriv ut ett tab-tecken"
    self queue nextPut: (String with: Character tab)
```

Initieringen av SharedQueue består i att tilldela variabeln queue en delad kö. Vi implementerar också en release-metod som tar bort den aktiva processen.

```
initialize
    queue := SharedQueue new.

release
    super release.
    process := nil.
    self show: ''.
    Instance := nil
```

Genom att tilldela process värdet nil så kommer villkoret för processens while-slinga inte vara uppfyllt. Processen kommer förmodligen ändå finnas kvar eftersom den oftast ligger och väntar på att kön ska fyllas på och den kommer troligen inte fyllas på mer eftersom instansen får meddelandet release. Så för att en sista gång använda kön och därigenom terminera processen skickar vi meddelandet show:'' till den.

KLASSMETODER

instance creation

```
new
    "Vi tillåter endast en instans av klassen och detaljerna för detta finns i
    metoden soleInstance"
    ^self soleInstance
```

```
    accessing
    soleInstance
        ^Instance
    class initialization
    initialize
        self soleInstance isNil ifTrue:[self initializeInstance]
    initializeInstance
        "skapar en ny instans av den delade kön och startar processen"
        Instance := super new initialize.
        Instance run
    displaying
    cr
        "skriv ut ny rad"
        self soleInstance cr
    show: aString
        "skriv ut en sträng"
        self soleInstance show: aString
    tab
        "skriv ut tab-tecken"
        self soleInstance tab
    examples
    twoProcesses
        "Två processer med olika prioritet som vi skriva ut i Transcript"
        [200 timesRepeat: [SharedTranscript show: 'a']] fork.
        [200
            timesRepeat:
                [SharedTranscript show: 'b'.
                (Delay forMilliseconds: 60) wait]]
                    forkAt: Processor userSchedulingPriority + 1
```

För att testa det hela kan vi helt enkelt använda exemplet, dvs utföra SharedTranscript twoProcesses.

Omstart av Image

Vad händer om Smalltalksystemet startas om efter det att det har sparats med processer som är aktiva? Svaret är helt enkelt att systemet ser till att dessa processer automatiskt startas om i det tillstånd de befann sig i då imagen sparades.

Säker utmatning med semafor

Det är också möjligt att använda en semafor för att direkt se till att utskrift i Transcript-fönstret fungerar även om många processer samtidigt skriver där. Det enklaste är att använda klassen Semaphore's kon-

struktion med kritisk sektion. Då har vi möjlighet att skydda de delar av koden som endast en process åt gången ska komma in i.

Vi subklassar TextCollector som är Transcript's klass.

```
TextCollector subclass: #SharedTextCollector
    instanceVariableNames: 'semaphore '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Bok-exempel'
```

Vi har lagt till en instansvariabel semaphore som håller i den instans av Semaphore som kommer se till att endast en process i taget skriver ut.

```
initialize
    semaphore := Semaphore forMutualExclusion.
    super initialize.
```

Instansvariabeln semaphore är avsedd att användas för kritiska sektioner. Som vanligt skriver vi en inspektor.

```
semaphore
    ^semaphore
```

Nu är det dags att kapsla in alla metoder för utskrift i en kritisk sektion. Den typiska konstruktionen blir att göra en kritisk sektion runt anropet till superklassens motsvarande metod.

```
accessing

nextPut: aCharacter
    "Skriv ett enstaka tecken på utmatningsbufferten."
    ^self semaphore critical: [super nextPut: aCharacter]

nextPutAll: aCollection
    "Skriv hela behållaren aCollection på utmatningsbufferten."
    ^self semaphore critical: [super nextPutAll: aCollection ]

cr
    "Radframmatning på utmatningsbufferten."
    ^self semaphore critical: [super cr]

crtab
    "Radframmatning följt av tabulering på utmatningsbufferten."
    ^self semaphore critical: [super crtab]

crtab: anInteger
    "En radframmatning och anInteger antal tabuleringstecken på
    utmatningsbufferten."
    ^self semaphore critical: [super crtab: anInteger]

space
    "Blanktecken på utmatningsbufferten."
    ^self semaphore critical: [super space]

tab
    "Tabulering på utmatningsbufferten."
    ^self semaphore critical: [super tab]
```

entry control

```
appendEntry
    "Töm utmatningsbuffertens innehåll på på en utanför objektet åtkomlig
    textbehållare (instansvariabeln value)."
    ^self semaphore critical: [super appendEntry]!

beginEntry
    "Töm utmatningbufferten."
    ^self semaphore critical: [super beginEntry]
```

Om vi tittar på hur metoderna ovan är definierade för superklassen TextCollector så ser vi att de direkt manipulerar datastrukturen som TextCollector bygger på utan att anropa några andra metoder i den egna klassen. Detta är viktigt då det kan uppkomma låsningsproblem om flera processer samtidigt utför olika kritiska sektioner som kontrolleras av samma semafor. När en process går in i den första kritiska sektionen låses alla andra processer från att gå in i kritiska sektioner som är kontrollerade av den aktuella semaforen. Men vad som kan vara ännu mera kritiskt är att också den körande processen hindras från att gå in andra kritiska sektioner som kontrolleras av samma semafor. Vad ska vi då göra? Vi måste se till att processer inte behöver gå in i ytterligare kritiska sektioner, tex genom viss kodduplicering.

Vi ska titta på ett exempel med metoden clear i TextCollector. Metoden är definierad på följande sätt där.

```
clear
    "Re-initialize the text to contain no characters."
    value := Text new.
    self beginEntry.
    self changed: #update
```

Först initieras instansvariabeln value till att innehålla en tom text. Därefter anropas metoden self beginEntry, som vi ju har skrivit om med kritisk sektion. Sist meddelas objekt som är beroende av den aktuella instansen av TextCollector att något har förändrats genom att ett changed-meddelande skickas.

Som första ansats kapslar vi in meddelandena i en kritisk sektion enligt:

```
SharedTextCollector>>clear
    "Gör textbehållare och utmatningsbuffert tomma."
    self semaphore critical: [
        value := Text new.
        self beginEntry.
        self changed: #update]
```

Med denna definition skyddar vi metoden clear, men eftersom clear anropar beginEntry som också har en kritisk sektion beroende av samma

semafor så kommer processen låsas vid det anropet. För att lösa dilemmat kopierar vi koden frånTextCollector>>beginEntry, dvs följande:

```
beginEntry
    "To speed up appending information to the receiver, a WriteStream is
    maintained.  Initialize it."
    entryStream := WriteStream on: (String new: 200)
```

Vi ser att det enda beginEntry gör är att initiera om instansvariabeln entryStream. Så vi kan helt enkelt härma denna initiering. Vi får då följande definition av clear.

```
clear
    self semaphore critical: [
        value := Text new.
        entryStream := WriteStream on: (String new: 200).
        self changed: #update]! !
```

Att göra på det här sättet är naturligtvis inte helt tillfredsställande eftersom vi inte längre utnyttjar arv och inkapsling i metoden. Nästa metod som ställer till problem är endEntry som anropar beginEntry.

Så här ser metoden ut i TextCollector:

```
endEntry
    "If the receiver's WriteStream is not empty, then reinitialize it.  Send all
    dependents a message that the streaming has changed."
    entryStream isEmpty
        ifFalse:
            [self changed: #appendEntry.
            self beginEntry]
```

Denna gång är vi också tvungna att eliminera anropet till den aktuella instansen eftersom det skulle låsa processen. Därför använder vi en annan teknik som minskar kodduplicieringen en aning, men som inte heller är speciellt snygg ur objektorienterad synvinkel.

```
endEntry
    ^self semaphore critical: [
        entryStream isEmpty
            ifFalse:
                [self changed: #appendEntry.
                super beginEntry]]
```

Vi kapslar in metoddefininitionen från superklassen inom en kritisk sektion precis som vår första ansats för clear. Men i detta fall utvecklar vi inte anropet till det egna objektet, dvs anropet beginEntry på sista raden utan vi anropar superklassens, dvs TextCollector beginEntry som inte är skyddad för samtidig åtkomst. Därigenom har vi undvikit att anropa en metod med kritisk sektion.

Som exempel skapar vi två processer med olika prioritet som samtidigt skriver på samma SharedTextCollector. Vi börjar först med att öppna ett fönster mot en instans av SharedTextCollector. Därefter skapar vi de båda processerna som skriver på den.

```
| transcript |
transcript := SharedTextCollector new.
TextCollectorView open: transcript label: 'Example transcript'.

[200 timesRepeat: [transcript show: 'a']] fork.
[200
    timesRepeat:
        [transcript show: 'b'.
        (Delay forMilliseconds: 15) wait]]
            forkAt: Processor userSchedulingPriority + 1
```

Sammanfattning

Termer

Lättviktsprocesser programdelar som kör oberoende av varandra fast inom samma adressrymd.

Semafor en systemsignal som ger möjlighet att synkronisera processer med varandra.

Kritisk sektion ett kodavsnitt som endast ska exekveras av en process åt gången.

Smalltalk nya klasser

Process beskriver ett program som kör, med dess programpekare och lokala variabler etc. Skapas från ett block.

ProcessScheduler har en instans som hanterar alla olika processer i systemet och vilken process som ska få köra vid ett visst tillfälle.

Semaphore kan signaleras mellan olika processer. En process kan vänta på att en signal ska signaleras av en annan process. Tex s := Semaphore new. [s wait. Transcript show: 'p1'] fork. [Transcript show: 'p2'. s signal] fork. En semafor kan också hantera kritiska sektioner.

Delay låter oss fördröja processer ett angivet antal millisekunder, sekunder eller tills systemklockan passerar ett viss klockslag. Tex d := Delay forMilliseconds: 100. d wait.

Smalltalk lättviktsprocesser

Skapas från block med meddelandet newProcess.

```
process := [kod] newProcess
```

Startas med resume

```
process resume
```

En process kan skapas och startas i ett steg med fork

```
process := [kod] fork
```

Ändra en process prioritet

```
process priority: ettHeltalsVärde
```

Konstanterna i ProcessorScheduler bör (ska) användas vid angivande av prioritet

```
process priority: Processor userBackgroundPriority
```

Meddelandet forkAt: kan användas för att direkt skapa och starta en process med given prioritet

```
process := block forkAt: Processor userBackgroundPriority
```

Övningar

13.1 Utgå från exemplet med klockan men utöka det stegvis med hjälp av subklassning

a) Låt klassen hantera processen som skriver ut tiden

b) Lägg till larmtider till klockan. För att få den att pipa använd `Screen default ringBell`

13.2 Skriv en klass som en gång i minuten kontrollerar om innehållet på en fil har ändrats. Om det har ändras läs in innhehållet i filen tolkat som ett smalltalkuttryck (`Object readFrom:`). Resultatet av uttrycket skrivs ut på annan fil. Baserat på denna enkla klass är det sedan möjligt att t ex ”fjärrstyra” en image.

13.3 Skriv en klass `Spyer`, som används för att följa exekveringen hos en målprocess av lägre prioritet. Låt `Spyer` samla in information om var i målprocessen mest tid spenderas (användbart för optimering).
Låt `Spyer` procecessen köra ca 20 gånger/sek och kontrollera då vad målprocessen gjorde innan den blev avbruten. För att ta reda på vad den andra processen gjorde använd meddelandet `suspendedContext` i `Process`. Omgivningen (kontexten) i sin tur håller information om metodnamn, mottagande objekt etc.

Ordlista

Abstrakt klass en klass som inte har några instanser. Används vanligen för att beskriva olika konkreta klassers gemensamma beteende och gränssnitt.

Adapter ett objekt som anpassar gränssnitt mellan olika objekt.

Aggregering ett objekt består av andra objekt som används för att uträtta arbetet, dvs helheten använder delarna.

Arv egenskaper i en klass kan användas i subklasser.

Attribut värden som hör till ett objekt.

Avlusare verktyg för att stegvis följa exekvering och meddelandesändning.

Behållare objekt som används för att lagra element.

Beroenden (eng. dependents) mellan objekt innebär att de kan deklarera intresse av andra objekts vitala förändringar. De andra objekten skickar sedan ut förändringsmeddelanden vid sådana förändringar.

Binärt meddelande ett meddelande med precis ett argument, t ex `2 + 3`.

Block en beskrivning av en fördröjd sekvens av operationer.

Browser fönsterbaserad programkodshanterare.

Brytpunkt kod som stoppar exekveringen och startar avlusaren. I Smalltalk mha meddelandet `halt`.

Del-av (eng. part-of) uttrycker att ett visst objekt är del av ett annat objekt, t ex en en motor är del-av en bil.

Dubbel uppslagning när ett metodanrop av generell typ specialiseras genom att skicka mottagaren som argument till argumentet.

Dynamisk bindning vilken metod som används avgörs av klasstillhörigheten vid exekveringen, inte vid kompileringen.

Ersättare (eng. proxy) ett objekt uppträder som ett annat objekt. Ofta för att på ett transparent sätt vidarebefodra eller filtrera meddelanden till andra objekt.

`false` speciell variabel som representerar konstanten falskt.

Förändringsmeddelanden (changed-meddelanden) kan skickas till ett objekt när vital del har ändrats. Detta resulterar i att eventuella beroende objekt meddelas om förändringen genom att uppdateringsmeddelanden skickas till dem.

Formell parameter representerar ett argument i en metod. Kan ej tilldelas nytt värde.

Generalisering en klass som är mer generell än sin superklass, t ex kan vi definiera Rektangel som subklass till Kvadrat.

Har-ett (eng. has-a) uttrycker att ett visst objekt har ett visst delobjekt, t ex en bil har-en motor.

Inkapsling att dölja ett objekts inre struktur från dess (externa) gränssnitt.

Inspektionsfönster speciellt fönster för att studera detaljer hos enskilda objekt.

Inspektor en metod som returnerar ett attribut.

Instans objekt skapat utifrån en klass. Har eget minnesutrymme.

Instansiering att skapa nya objekt från en klass.

Instansvariabel en del av ett objekts privata minne.

Instansvariabel i klass fungerar analogt med "vanliga" instansvariabler, dvs lokalt per objekt (här klass).

Iteration eller *upprepning* med whileTrue:, to:do: eller to:do:by:. Ett blockargument beskriver vad som ska göras i slingan.

Jo-jo-problemet beskriver svårigheten att följa metoduppslagning om flera metoder i en klasshierarki används omvartannat.

Kaskadmeddelande ett sätt att på ett enkelt sätt skicka flera meddelanden till samma objekt.

Klass beskrivning av ett objekt med data och operationer på dessa data.

Klassvariabel en variabel som delas mellan en klass, alla subklasser till den samt alla instanser av klassen och subklasserna.

Konkret klass en klass som kan instansieras.

Konstruktör en klassmetod som används för att instansiera klassen.

Kritisk sektion ett kodavsnitt som endast ska exekveras av en process åt gången.

Länkad lista behållare som använder pekare till elementen.

Lat initiering ett attribut binds till skönsvärde först vid första anropet av dess inspektor.

Lättviktsprocesser programdelar som kör oberoende av varandra fast inom samma adressrymd.

Litteral form ett sätt att beskriva ett objekt direkt utan att skapa det via ett meddelande till en klass.

Litteral konstant objekt som kan skapas direkt med speciella syntaktiska konstruktioner.
Meddelande en uppmaning till ett objekt att utföra en metod.
Metaklass objekt vars instanser är klasser.
Metod en beskrivning av hur ett visst objekt ska utföra en av sina operationer. Motsvarande procedur, funktion, subrutin i andra språk.
Metodkatalog en klass metoder finns i en katalog med metodnamnen som nycklar och koden som värden.
Metodnamn en metods namn, exklusive parametrar. T ex abs, transactions, +, min:, beteween:and: eller move:to:.
Metoduppslagning att slå upp en metod i metodkatalogen.
Mottagare det objekt som uppmanas att utföra en metod.
Multipelt arv arv från flera superklasser, ej möjligt i Smalltalk.
Mutator en metod som direkt ändrar ett attribut och inget annat.
nil anger det värde som en variabel har om inget speciellt objekt har tilldelats det. Alla variabler har detta värde från början.
Nyckelord en identifierare med ett avslutande kolon, t ex nyckel:.
Nyckelordsmeddelande ett meddelande med ett eller flera argument åtskilda med nyckelord. T ex x between: 3 and: 4.
Objekt en komponent i systemet med eget privat minne och en uppsättning operationer.
OMT Object Modeling Technique en analys- och designmetod vars syntax används i figurerna i boken.
Polymorfi olika metoder kan ha samma namn.
Poolvariabel variabel som delas mellan klasser som inte (nödvändigtvis) befinner sig i samma arvshierarki.
Prefixklass synonymt med superklass.
Primitiv metod en metod som utförs direkt av den virtuella maskinen.
Pseudovariabel variabel vars värde inte kan påverkas av programmet. Variablerna self, super, true och nil är exempel på sådana.
Rotklass klassen högst upp i arvsträdet.
Selektor synonymt med metodnamn. Ibland avses metodnamnets symboliska form, t ex #between:and:.
self en speciell variabel som refererar till det objekt som utför en metod, dvs mottagaren av motsvarande meddelande.
Semafor en systemsignal som ger möjlighet att synkronisera processer med varandra.

Skräpsamling systemet städar automatiskt bort objekt som ej längre refereras.

Spårutskrifter programsatser med utmatning i utskriftsfönstret för att följa exekveringen.

Specialisering en klass som är mer specialicerad än sin superklass, t ex kan vi definiera Cirkel som subklass till Ellips.

Speciell variabel se pseudovariabel.

Statisk bindning vilken metod som ska användas avgörs redan vid kompileringen.

Subklass (härledd klass) en klass som ärver variabler och egenskaper från en existerande klass.

super en speciell variabel som precis som self refererar till det objekt som utför en viss metod. Indikerar att metod vid meddelande till objektet ska börja sökas i klasser högre upp i klasshierarkin.

Superklass den klass varifrån variabler och egenskaper ärvs.

Temporär variabel en variabel som konstrueras för att mellanlagra värden under en viss sekvens av operationer. Deklareras mellan ett par av vertikala linjer. Exempel: | temp1 temp2 |.

thisContext refererar aktuell exekveringsomgivning.

Transcript se utskriftsfönster.

true speciell variabel som representerar konstanten sant.

Unärt meddelande ett meddelande utan argument, t ex 0.5 cos.

Uppdateringsmeddelanden (update-meddelanden) som resultat av att ett objekt skickar ett förändringsmeddelande får alla beroende objekt ett uppdateringsmeddelande som informerar om vad och vem som har förändrats.

Utskriftsfönster (eng Transcript) fönster för utskrift av systemmeddelanden och för egna utskrifter.

Val skrivs som villkor följt av ifTrue: eller ifFalse: där argumentet är ett block som beskriver vad som skall göras om villkoret är uppfyllt. T ex x >= y ifTrue: ['x störst'].

Värdehållare (eng. value holder) ett objekt som håller i ett värde. Meddelar beroende objekt om värdet förändras.

Variabel är temporär, global, instans-, klass- eller poolvariabel. Får värdet nil från början.

Variabel subklass klass med indexerbara variabler. Finns två olika typer nämligen: - med variabler av pekartyp - med variabler av icke

pekartyp där elementen är byte.

Virtuell maskin (objektsmaskin) maskinberoende del av smalltalksystemet som hanterar exekvering av kod.

Virtuella metoder metoder vars användning bestäms först vid exekveringen, inte kompileringen. Motsvarar dynamisk bindning.

Åtkomstmetod är antingen en inspektor eller mutator.

Är-en (eng. is-a) uttrycker att ett objekt är av en viss sort, t ex en bil är-en farkost.

Referenslista

Analys, design och modellering

[1] Booch *Object-Oriented Design* The Benjamin/Cummings Publishing Company, Inc. 1991.

[2] Rumbaugh, Blaha, Premerlani, Eddi och Lorensen *Object-Oriented Modelling and Design* Prentice Hall 1991.

Objektorienterad programmering

Allmänna

[3] Budd *An Introduction to Object-Oriented Programming*, Addison Wesley 1991.

[4] Cox *Object Oriented Programming an Evalutionary Approach* Addison Wesley 1986.

[5] Masini, Napoli, Colnet, Leonard och Tombre O*bject Oriented Languages* Academic Press 1991.

[6] Meyer *Object-oriented Software Construction* Prentice Hall 1988.

Språk

[7] Dahl och Lindqvist *Objektorienterad programmering och algoritmer i Simula*, Studentlitteratur 1993.

[8] Goldberg och Robson *Smalltalk-80 The Language and it's Implementation* Addison Wesley 1983.

[9] LaLonde *Discovering Smalltalk* The Benjamin/Cummings Publishing Company, Inc. 1994.

[10] LaLonde och Pugh *Inside Smalltalk volume I* Prentice Hall 1991.

[11] LaLonde och Pugh *Smalltalk V* Prentice Hall 1994.

[12] Strousrup *The C++ Programming Language* Addison Wesley 1991.

[13] Sundblad, Romberger och Leringe *Fortsatt Programmering i SIMULA* Teknisk Högskolelitteratur i Stockholm AB 1981.

Algoritmer och analys

[14] Aho, Hopcroft och Ullman *The Design and Analysis of Computer Algorithms* Addison Wesley 1974.

Numerisk analys

[15] Björk och Dahlquist *Numerical Methods* Prentice hall 1974.

Interaktionsprogrammering och system

[16] Eiderbäck, Hägglund och Bälter *Smalltalkmiljön i VisualWorks.* Kommande på Studentlitteratur.

[17] Eiderbäck, Hägglund och Bälter *Interaktionsprogrammering i Smalltalk.* Kommande på Studentlitteratur.

[18] Goldberg *Smalltalk-80 The Interactive Programming Environment* Addison Wesley 1983.

[19] Krasner och Pope *Cookbook for using the Model-View-Controller User Interface Paradigm*, Journal of Object Oriented Programming, August 1988, s. 26-49.

Programmering allmänt

[20] Abelson och Sussman *Structure and Interpretation of Computer Programs* MIT Press 1985.

[21] Sethi *Programming Languages* Addison Wesley 1989.

Diverse

[22] Klarner (editor) *The Mathematical Gardner* Wadsworth International 1981.

Lösningsförslag

Kapitel 1

1.5

```
a) (a min: b) min: c
b) a <= b
        ifTrue: [a <= c
            ifTrue: [min := #a]
            ifFalse: [min := #c]]
        ifFalse: [b <= c
            ifTrue: [min := #b]
            ifFalse: [min := #c]].
    min
```

1.6

```
    | start stop |
    start := 37.
    stop := 42.
    start to: stop do: [:c | Transcript show: c printString, 'c = '.
                        Transcript show: (9 / 5 * c + 32.0) printString, 'f'.
                        Transcript cr]
```

1.7

```
    1 to: 10 do: [:x |
        1 to: 10 do: [:y | Transcript show: (x * y) printString.
                    Transcript tab].
        Transcript cr]
```

1.8

```
    | term e epsilon n |
    n := 1.
    term := 1.
    e := 1.0.
    epsilon := 0.001.
    [term > epsilon]
        whileTrue: [term := term / n.
                    n := n + 1.
                    e := e + term].
    e
```

$\Longrightarrow$ *2.71825*

1.9

```
a) 2 + 3
```

b) 2 * 3 + 4
c) 2 + (3 * 4)
d) 10 + (5/2 * 3/4)
e) skriv något tal som flyttal tex 10.0 eller använd meddelandet asFloat

1.10

a)

```
| sum a b |
    sum := 0.
    a := 5.
    b := 7.
    a to: b do: [:i | sum := sum + i].
    sum
```

b)

```
| sum a b count |
    sum := 0.
    a := 5.
    b := 7.
    count := a.
    [count <= b]
        whileTrue: [sum := sum + count.
                    count := count + 1].
    sum
```

c) a + b * (b - a + 1) / 2
d) (a to: b) inject: 0 into: [:x :y | x + y]

Kapitel 2

2.1

```
widthdraw: amount
    self balance: self balance - amount
```

2.2

```
surname
    ^surname

surname: aString
    surname := aString
```

2.3

```
Object subclass: #BodyMassIndex
    instanceVariableNames: 'vikt längd '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Book-exerc'

längd
    ^längd

längd: meter
    längd := meter
```

```
vikt
    ^vikt

vikt: kg
    vikt := kg

bmi
    ^self vikt / self längd squared

"test"
| body |
body := BodyMassIndex new.
body längd: 1.80.
body vikt: 96.
body bmi
```

⟹ *29.6296*

2.4

Rekursiv variant:

```
tjebysjev: x
    self = 0 ifTrue: [^1].
    self = 1 ifTrue: [^x].
    ^2 * x * (self - 1 tjebysjev: x) - (self - 2 tjebysjev: x)
```

Iterativ version:

```
tjebysjevIter: x
    | t0 t1 twoX |
    self = 0 ifTrue: [^1].
    self = 1 ifTrue: [^x].
    t0 := 1.
    t1 := x.
    twoX := 2 * x.
    self - 1
        timesRepeat:
            [| slask |
            slask := t1.
            t1 := twoX * t1 - t0.
            t0 := slask].
    ^t1
```

2.5

```
Object subclass: #Student
    instanceVariableNames: 'förnamn efternamn utbildningslinje '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Kap2-exercise'

efternamn
    ^efternamn

efternamn: aString
    efternamn := aString

förnamn
    ^förnamn
```

```
förnamn: aString
    förnamn := aString

utbildningslinje
    ^utbildningslinje

utbildningslinje: aString
    utbildningslinje := aString
```

2.6

a)

```
Object subclass: #Artikel
    instanceVariableNames: 'artikelnummer artikel namn antalILager antalSålda
                            pris '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Kap2-exercise'
```

b)

```
lagerVärde
    ^self antalILager * self pris

försäljningsIntäkter
    ^self antalSålda * self pris
```

2.7

a)

```
Object subclass: #Punkt
        instanceVariableNames: 'x y '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Egen-Geometri'

x
    ^x

x: aNumber
    x := aNumber

y
    ^y

y: aNumber
    y := aNumber
```

b)

```
dist: aPunkt
    ^((self x - aPunkt x) squared
        + (self y - aPunkt y) squared) sqrt
```

c)

```
translatedBy: enPunkt
    | nyPunkt |
    nyPunkt := Punkt new.
    nyPunkt x: self x + enPunkt x.
    nyPunkt y: self y + enPunkt y.
    ^nyPunkt
```

d)

```
scalededBy: amount
    | nyPunkt |
    nyPunkt := Punkt new.
    nyPunkt x: self x * amount.
    nyPunkt y: self y * amount.
    ^nyPunkt
```

2.8

a)

```
Object subclass: #Rektangel
    instanceVariableNames: 'origin extent '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Egen-Geometri'
```

```
extent
    ^extent
```

```
extent: aValue
    extent := aValue
```

```
origin
    ^origin
```

```
origin: aValue
    origin := aValue
```

b)

```
extent: enPunkt
    | rect |
    rect := Rektangel new.
    rect origin: self.
    rect extent: enPunkt.
    ^rect
```

c)

```
area
    ^self extent x * self extent y
```

d)

```
center
    | punkt |
    punkt := Punkt new.
    punkt x: self origin x + (self extent x / 2).
    punkt y: self origin y + (self extent y / 2).
    ^punkt
```

e)

```
moveBy: enPunkt
    self origin x: self origin x + enPunkt x.
    self origin y: self origin y + enPunkt y
```

f)

Skriv metod postCopy så funkar Object>>copy direkt

```
postCopy
    origin := origin copy.
    extent := extent copy
```

Kapitel 3

3.1

a) Fraction
b) UndefinedObject
c) UndefinedObject

3.2

a)

```
(2/3) isMemberOf: Fraction
```

⇒*true*

b)

```
| x |
x := 'en sträng'.
x isMemberOf: Float
```

⇒*false*

3.3

a)

```
Fraction superclass
```

⇒Number

b)

```
4 class
```

⇒*SmallInteger*

3.4

a)

```
Number subclasses
```

⇒*#(FixedPoint Fraction Integer LimitedPrecisionReal)*

b)

```
#('test') class superclass
```

⇒*ArrayedCollection*

3.5

```
christianName: aString
    christianName := aString

familyName
    ^familyName
```

3.6

Metoden name använder meddelandet assureName så om vi skriver self name i assureName får vi oändlig rekursion.

3.7

```
| eps solve func |
    eps := 0.001.
    solve := [:f :a :b || m v |
        m := a + b / 2.
        v := f value: m.
        v abs < eps
            ifTrue: [m]
            ifFalse:
                [v < 0
                    ifTrue: [solve value: f value: m value: b]
                    ifFalse: [solve value: f value: a value: m]]].
    func := [:x | x sin].
    solve value: func value: -1 + eps value: 1 - eps
```

3.8 a) 127 b) 123 c) 128

Kapitel 4

4.5

```
Person subclass: #Student
    instanceVariableNames: 'utbildningslinje '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Kap4-exercise'
```

INSTANSMETODER

```
utbildningslinje
    ^utbildningslinje

utbildningslinje: aString
    utbildningslinje := aString
```

KLASSMETODER

```
förnamn: enBehållare efternamn: enSträng utbildningslinje:
enAnnanSträng
    ^(self christianNames: enBehållare familyName: enSträng)
        utbildningslinje: enAnnanSträng
```

4.6

```
Object subclass: #Student
    instanceVariableNames: 'person utbildningslinje '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Kap4-exercise'
```

INSTANSMETODER

```
förnamn: enBehållare efternamn: enSträng
    self person: (Person christianNames: enBehållare familyName: enSträng)

person
    ^person
```

person: aPerson
person := aPerson

efternamn
^self person surname

efternamn: aString
self person surname: aString
"Övriga åtkomstmetoder pss"

KLASSMETODER

new
^self shouldNotImplement

förnamn: enBehållare efternamn: enSträng utbildningslinje: enAnnanSträng
^(self basicNew förnamn: enBehållare efternamn: enSträng)
utbildningslinje: enAnnanSträng

4.9

a)

Lite lyxigare än uppgiften kräver (d-uppgiften inbakad)

```
Object subclass: #Turtle
    instanceVariableNames: 'position direction '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Turtles'
```

INSTANSMETODER

initialize-release

initialize
self initializeLocation: Point zero.
self direction: 90 "Face down!"

accessing

direction
^direction

direction: anAngle
direction := anAngle \\ 360 "modulo 360"

position
^position

position: aPoint
self setPosition: aPoint

moving

go: distance
| dir scaleFactor |
dir := self direction degreesToRadians.
scaleFactor := dir cos @ dir sin.
self position: scaleFactor * distance + self position

point functions

dist: anotherTurtle
^self position dist: anotherTurtle position

transforming

turn: angle
^self direction: self direction + angle

private

initializeLocation: newPosition
position := newPosition

KLASSMETODER

instance-creation

new
^self basicNew initialize
"alternativelly
^super new initialize"

position: initialPosition
^self new initializeLocation: initialPosition

position: initialPosition direction: initialAngle
| newSelf |
newSelf := self new.
newSelf initializeLocation: initialPosition.
newSelf direction: initialAngle.
^newSelf

b)

```
Turtle subclass: #LogTurtle
    instanceVariableNames: 'distanceTravelled '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Turtles'
```

INSTANSMETODER

accessing

distanceTravelled
^distanceTravelled

distanceTravelled: newValue
distanceTravelled := newValue

position: aPoint
self distanceTravelled: self distanceTravelled + (self position dist: aPoint).
super position: aPoint

private

initializeLocation: newPosition
super initializeLocation: newPosition.
self distanceTravelled: Number zero

c)

```
LogTurtle subclass: #LyxTurtle
    instanceVariableNames: 'startPosition areaAcc '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Turtles'
```

INSTANSMETODER

accessing

```
area
    ^areaAcc +
        (self position x * self startPosition y -
            (self startPosition y * self startPosition x) / 2)
```

```
position: aPoint
    | oldP |
    oldP := self position.
    super position: aPoint.
    areaAcc := areaAcc + (oldP x * aPoint y - (aPoint x * oldP y) / 2)
```

```
startPosition
    ^startPosition
```

private

```
initializeLocation: newPosition
    super initializeLocation: newPosition.
    startPosition := self position.
    areaAcc := Number zero
```

Kapitel 5

5.1 svar: 1

5.2

a)

```
fibIter
    | f0 f1 |
    f0 := f1 := 1.
    self - 1 timesRepeat:
        [| fOld |
        fOld := f1.
        f1 := f1 + f0.
        f0 := fOld].
    ^f1
```

b)

```
fibDirectly
    | root |
    root := 5 sqrt.
    ^1 + root / 2 ** (self + 1) - (1 - root / 2 ** (self + 1)) / root
```

5.3

```
isPrime
    | root current |
    self even ifTrue: [^self = 2].
    root := self sqrt.
    current := 3.
    [current < root]
        whileTrue:
            [self \\ current = 0 ifTrue: [^false].
            current := current + 2].
    ^true
```

5.5 Byt ut villkoret mot self between: epsilon and: m - epsilon där epsilon något litet värde.

Kapitel 6

6.2

a)

```
fac := [:n | n <= 1
        ifTrue: [1]
        ifFalse: [n * (fac value: n -1)]]
```

b)

```
fac:= [:n | (1 to: n) inject: 1 into: [:x :y | x * y]]
```

6.3

```
| isPrime |
isPrime := [:n |
    n even | (n < 2)
        ifTrue: [n = 2]
        ifFalse:
            [3 to: n sqrt truncated do:
                [:i | n \\ i = 0 ifTrue: [^false]].
                true]].
isPrime value: 4711
```

6.4

```
DestinationClass copyAllCategoriesFrom: SourceClass.
DestinationClass class copyAllCategoriesFrom: SourceClass class.
```

6.5

```
Object subclass: #TimeProxy
    instanceVariableNames: 'object time '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Ersättare'
```

INSTANSMETODER

prototypes

```
doesNotUnderstand: aMessage
    | ans |
    time := time + (Time millisecondsToRun: [ans := self theEncapsulatedObject
    perform: aMessage selector withArguments: aMessage arguments]).
    ^ans
```

initialization-release

```
initialize
    time := 0
```

accessing'

```
proxyTime
    ^ time
```

```
restoreTheEncapsulatedObjectAndReturnTime
    |  theTime |
    theTime := time.
    self become: self theEncapsulatedObject.
    ^theTime
```

```
theEncapsulatedObject
    ^object
```

```
theEncapsulatedObject: anObject
    object := anObject shallowCopy.
    anObject become: self
```

KLASSMETODER

instance creation

```
new
    ^super new initialize
```

```
on: anObject
    ^self new theEncapsulatedObject: anObject
```

special initialization

```
initialize
    self == TimeProxy
        ifTrue:
            [superclass := nil.
            #(  #class #basicInspect #basicSize #become: #perform:
                #perform:with: #perform:with:with: #perform:with:with:with:
                #performWithArguments: #performMethod: #performMethod:with:
                #performMethod:with:with: #performMethod:with:with:with:
                #performMethod:arguments:)
                        do: [:selector | self copy: selector from: Object]]
```

När koden är inläst måste klassen initieras med hjälp av:

```
TimeProxy initialize
```

6.7

```
initsubclass: className instanceVariableNames: instVarsAndInitialization
classVariableNames: classVariableNames poolDictionaries: pools
category: category
    "skapar klass med mutatorer och initieringsmetoder"
    | newClass  instanceVariableNames |
    instanceVariableNames := ''.
    1 to: instVarsAndInitialization size by: 2 do: [:i |
        instanceVariableNames := instanceVariableNames , ' ' ,
                                        (instVarsAndInitialization at: i) asString].
    newClass := self
            recordsubclass: className
            instanceVariableNames: instanceVariableNames
            classVariableNames: classVariableNames
            poolDictionaries: pools
            category: category.
    newClass  createMethodNew.
    newClass createInitializeMethodOf: instVarsAndInitialization

createMethodNew
    self class compile: 'new
    ^self basicNew initialize' classified: #'instance creation'

createInitializeMethodOf: instVarsAndInitialization
    "Genererar intitieringsmetod för mottagande klass"
    | methodString |
    methodString := 'initialize
    "denna initieringsmetod är automatgenererad"
    super initialize.
    '.
    1 to: instVarsAndInitialization size by: 2do: [:i |
            methodString := methodString , '' ,
                                (instVarsAndInitialization at: i) asString, ' := ' ,
                                        (instVarsAndInitialization at: i + 1) asString, '.
    '].
    self compile: methodString classified: #'initialize-release'
```

Kapitel 7

7.6

```
| intAux step int |
intAux := [:f :a :b :h |
        h * ((a + h to: b - h by: h)
                inject: (f value: a)
                        + (f value: b) / 2
                into: [:subTotal :next | subTotal +  (f value: next)])].
step := 0.01.
int := [:f :a :b |
         intAux valueWithArguments: (Array with: f with: a with: b with:step)].
```

Vi prövar med den föreslagna funktionen över intervallet [0, 1], dvs vi beräknar $\int_0^1 \frac{dx}{1+x}$.

```
int
    value: [:x | 1 / (1 + x)]
    value: 0
    value: 1
```

⟹*0.693153*

[Det analytiska värdet är ln(2) - ln(1) = ln(2) ≈ 0.693147181 (Vi använde här steglängden 0.01, men om vi kortar den hamnar vi närmare det analytiska värdet. Fast om steget blir för kort kan beräkningarna ta lång tid eller vissa numeriska problem uppstå (se standardtext i Numerisk Analys))]

7.7

```
| pattern guess patternCopy elementNo black  white |
"pattern och guess är testmönster"
pattern := #(1 2 3 4 5 3).
guess := #(2 1 3 6 1 4) asOrderedCollection.
"Vi skapar kopia så att mönstret inte förstörs (om B behöver göra flera
gissningar)"
patternCopy := pattern asOrderedCollection.
elementNo := 1.
black := 0.
"Nu tar vi bort alla exakta matchningar"
[elementNo <= patternCopy size]
    whileTrue: [(patternCopy at: elementNo) = (guess at: elementNo)
                ifTrue: [black := black + 1.
                        patternCopy removeAtIndex: elementNo.
                        guess removeAtIndex: elementNo]
                ifFalse: [elementNo := elementNo + 1]].
white := 0.
"Det som är kvar kan ej vara exakta matchningar"
patternCopy do:[:x | (guess includes: x)
                    ifTrue: [white := white + 1.
                            guess remove: x]]
```

Kapitel 8

8.1

```
do: aBlock
    | i |
    i := 0.
    [(i := i + 1) < self size] whileTrue: [aBlock value: (self at: i)]
```

Kapitel 10

10.5

```
ReadStream subclass: #SmartStream
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Book-exerc'
```

INSTANSMETODER

acccessing

next

```
    [| nxt |
    nxt := super next.
    nxt isNil ifTrue: [^nxt].
    nxt isAlphaNumeric ifTrue: [^nxt asUppercase]] repeat
```

10.6

Enklast möjliga lösning. Går enkelt att förbättra/optimera vid behov! Dvs vi kan skriva om klassen så att den verkligen går baklänges och inte gör den onödiga operationen att vända objektet.

```
SmartStream subclass: #SmartStreamBackward
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Book-exerc'
```

KLASSMETODER

instance creation

on: aCollection

```
    ^super on: aCollection reverse
```

10.7

För att göra det enkelt för oss använder vi SmartStream och SmartStream-Backward från tidigare övningar.

isPalindrom

```
    | forward backward mySize |
    forward := SmartStream on: self.
    backward := SmartStreamBackward on: self.
    mySize := self size.
    [forward position > (mySize - backward position)]
        whileFalse: [forward next ~= backward next ifTrue: [^false]].
    ^true
```

10.8

Vi kan skriva flera olika indexord mellan märkena (@). Sedan är det enkelt att stycka upp detta resultat.

10.9

```
| tecken rad slump |
tecken := #($1 $x $2).
rad := Array new: 13.
slump := Random new.
1 to: rad size do:
    [:i | rad at: i
            put: (tecken at: (slump next * 3 + 1) truncated)].
rad
```

10.10

Vi inför en ny metod doPrintOn: i var och en av klasserna. Denna metod beskriver vad som är speciellt för en viss typ av sköldpadda. Det som är gemensamt samt anropet av den nya metoden skriver vi i printOn:.

```
Turtle>>printOn: aStream
    aStream nextPutAll: self class name; cr; nextPutAll: ' ('.
    self doPrintOn: aStream.
    aStream nextPut: $)

Turtle>>doPrintOn: aStream
    aStream nextPutAll: 'position = '.
    self position printOn: aStream.
    aStream nextPutAll: ' direction = '.
    self direction printOn: aStream

LogTurtle>>doPrintOn: aStream
    super doPrintOn: aStream.
    aStream nextPutAll: ' distance = '.
    self distanceTravelled printOn: aStream.

LyxTurtle>>doPrintOn: aStream
    super doPrintOn: aStream.
    aStream nextPutAll: ' parea = '.
    self area printOn: aStream
```

Kapitel 11

11.1

a) Filename named: 'MinFil'
b. 'MinFil' asFilename

11.5

Vissa kontroller men minimalt med felmeddelanden. Vi visar också hur en enkel användardialog där användaren frågas efter ett filnamn kan konstrueras.

```
| chooice |
"Fråga efter fil"
(chooice := Dialog requestFileName: 'Summera filen:' default: '*') isEmpty
    ifFalse: [| file | "Svaret var inte den tomma strängen"
        file := chooice asFilename.
```



```
        "Får vi läsa filen?"
        file isReadable
            ifTrue: [| stream sum |
                sum := 0.
                stream := file readStream.
                "Nu löser vi den egentliga uppgiften"
                [[stream atEnd]
                    whileFalse: [stream peek isDigit
                                    ifTrue: [sum := sum + (Number readFrom:
                                                                    stream)]
                                    ifFalse: [stream skip: 1]]]
                            valueNowOrOnUnwindDo: [stream close].
                sum]]
```

11.6

```
| chooice |
(chooice := Dialog requestFileName: 'Konvertera fil:' default: '*') isEmpty
    ifFalse: [| file |
        file := chooice asFilename.
        "Får vi skriva på filen?"
        file canBeWritten
            ifTrue: [| stream |
                stream := file readWriteStream.
                "Nu löser vi den egentliga uppgiften"
                [[stream atEnd]
                    whileFalse: [stream nextPut: stream peek asUppercase]]
                        valueNowOrOnUnwindDo: [stream close]]]
```

11.7

```
| chooice |
(chooice := Dialog requestFileName: 'Konvertera fil:' default: '*') isEmpty
    ifFalse: [| file |
        file := chooice asFilename.
        "Får vi skriva på filen?"
        file canBeWritten
            ifTrue: [| stream replacements |
                replacements := Dictionary new.
                replacements at: $} put: $å.
                replacements at: ${ put: $ä.
                replacements at: $| put: $ö.
                replacements at: $] put: $Å.
                replacements at: $[ put: $Ä.
                replacements at: $\ put: $Ö.

                stream := file readWriteStream.
                [[stream atEnd]
                    whileFalse: [| nxt |
                            nxt := stream peek.
                            (replacements includesKey: nxt)
                                ifTrue: [stream nextPut: (replacements at: nxt)]
                                ifFalse: [stream skip: 1]]]
                            valueNowOrOnUnwindDo: [stream close]]]
```

alternativt kan

```
(replacements includesKey: nxt)
    ifTrue: [stream nextPut: (replacements at: nxt)]
    ifFalse: [stream skip: 1]
```

ersättas med

```
stream nextPut: (replacements at: nxt ifAbsent: [nxt])
```

11.13

```
| nameOfFile |
(nameOfFile := Dialog
                    requestFileName: 'Name a file'
                    default: '*.st') isEmpty
    ifFalse:
        [| file |
        file := nameOfFile asFilename.
        (file isReadable)
            ifTrue:
                [| stream bag  |
                stream := file readStream.
                bag := Bag new.
                [stream atEnd] whileFalse: [| next wordStream word |
                    wordStream := '' writeStream.
                    [next := stream next.
                    next isLetter] whileTrue: [wordStream nextPut: next].
                    (word := wordStream contents) isEmpty
                        ifFalse: [bag add: word]].
                stream close.
                bag]]
```

ger exempelvis (med en fil med den aktuella koden som indata):

⇒*Bag ('new' 'file' 'file' 'file' 'file' 'file' 'nameOfFile' 'nameOfFile' 'nameOfFile' 'isEmpty' 'isEmpty' 'Name' 'writeStream' 'ifFalse' 'ifFalse' 'wordStream' 'wordStream' 'wordStream' 'wordStream' 'next' 'next' 'next' 'next' 'next' 'nextPut' 'contents' 'isReadable' 'Dialog' 'close' 'Bag' 'stream' 'stream' 'stream' 'stream' 'stream' 'st' 'requestFileName' 'asFilename' 'default' 'a' 'readStream' 'word' 'word' 'word' 'isLetter' 'whileTrue' 'atEnd' 'bag' 'bag' 'bag' 'bag' 'whileFalse' 'add' 'ifTrue')*

Behandling av resultatet:

```
| collection |
collection := SortedCollection sortBlock: [:x :y | x key <= y key].
bag valuesAndCountsDo: [:v :c | collection add: v->c].
collection
```

⇒*SortedCollection ('a'->1 'add'->1 'asFilename'->1 'atEnd'->1 'Bag'->1 'bag'->4 'close'->1 'contents'->1 'default'->1 'Dialog'->1 'file'->5 'ifFalse'->2 'ifTrue'->1 'isEmpty'->2 'isLetter'->1 'isReadable'->1 'Name'->1 'nameOfFile'->3 'new'->1 'next'->5 'nextPut'->1 'readStream'->1 'requestFileName'->1 'st'->1 'stream'->5 'whileFalse'->1 'whileTrue'->1 'word'->3 'wordStream'->4 'writeStream'->1)*

13.2

```
Object subclass: #FileReader
    instanceVariableNames: 'inFile outFile lastCheckTime process '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Book-Examples'
```

INSTANSMETODER

initialize-release

```
initialize
    self lastCheckTime: Time dateAndTimeNow

release
    super release.
    self process: nil.
```

accessing

```
inFile
    ^inFile

inFile: newValue
    inFile := newValue

inFile: anInFile1 outFile: anOutFile2
    self inFile: anInFile1.
    self outFile: anOutFile2.

lastCheckTime
    ^lastCheckTime

lastCheckTime: newValue
    lastCheckTime := newValue

outFile
    ^outFile

outFile: newValue
    outFile := newValue

process
    ^process

process: aProcess
    process := aProcess
```

private

```
readFromFile
    "läser från infilen och tolkar det som ett Smalltalk
    uttryck. Skriver ut resultatet på outFile"
    | inStrm ans outStrm |
    inStrm := self inFile asFilename readStream.
    [ans := Object readFrom: inStrm]
        valueNowOrOnUnwindDo: [inStrm close].
    outStrm := self outFile asFilename writeStream.
    [outStrm nextPutAll: ans printString]
        valueNowOrOnUnwindDo: [outStrm close]
```

```
readIfChanged
    | changedTime |
    changedTime := self inFile asFilename dates at: #modified.
    ((changedTime at: 1)
        > (self lastCheckTime at: 1) or: [(changedTime at: 1)
            = (self lastCheckTime at: 1) and: [(changedTime at: 2)
                > (self lastCheckTime at: 2)]])
        ifTrue:
            [self lastCheckTime: changedTime.
            self readFromFile]
```

process controlling

```
run
    "startar processen som kollar om infilen har ändrats"
    self process: ([[self process == Processor activeProcess]
            whileTrue: [(Delay forSeconds: 60) wait.
                    self readIfChanged]]
        forkAt: Processor userBackgroundPriority)
```

KLASSMETODER

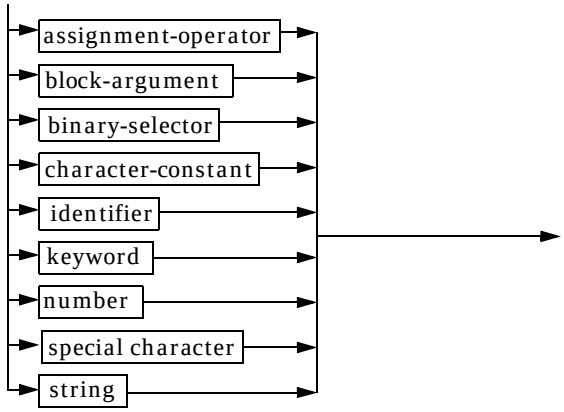
instance creation

```
inFile: anInFile1 outFile: anOutFile2
    "self inFile: 'in' outFile: 'ut'"
    ^self new inFile: anInFile1 outFile: anOutFile2

new
    ^super new  initialize
```

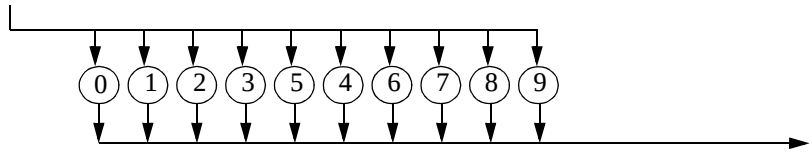
Syntaxdiagram

Character classes

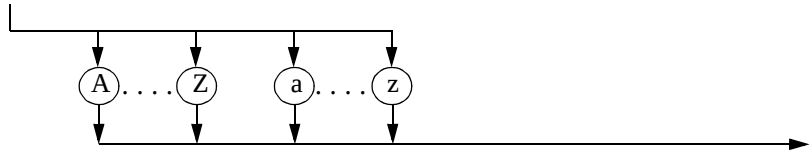
token



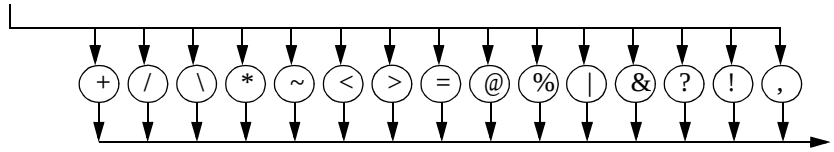
digit



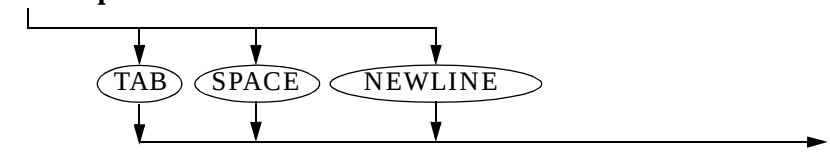
letter



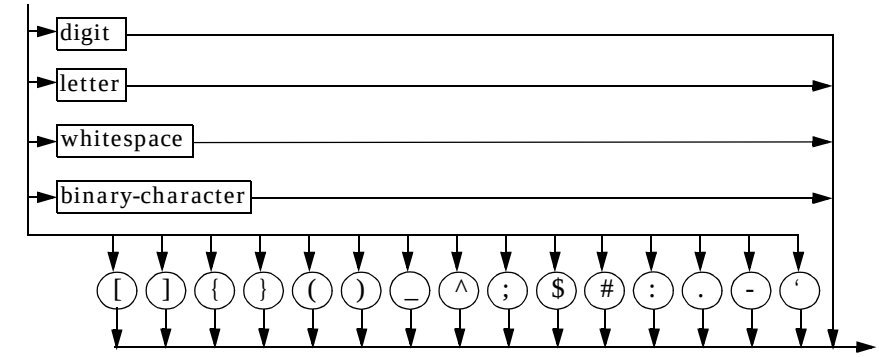
binary-character



whitespace

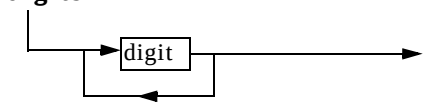


non-quote-character

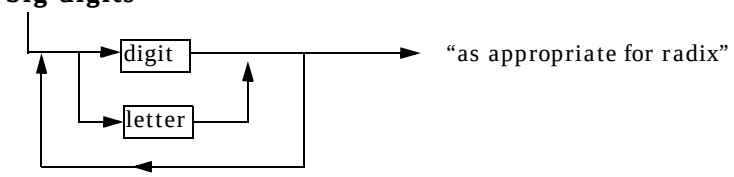


Numbers

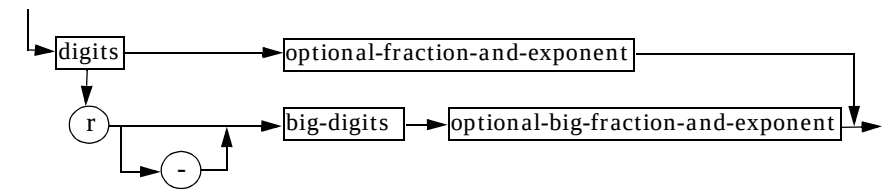
digits



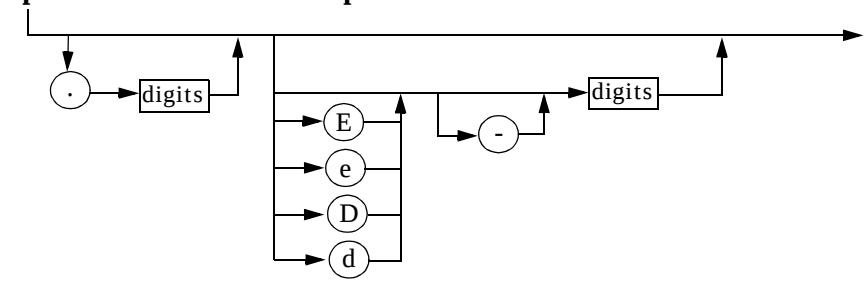
big-digits



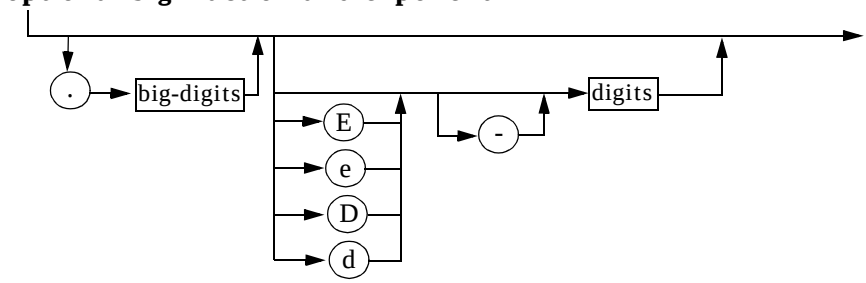
number



optional-fraction-and-exponent

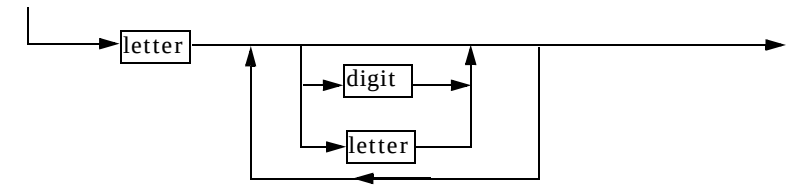


optional-big-fraction-and-exponent

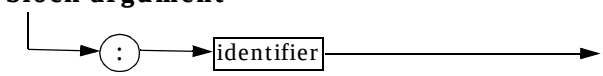


Other lexical constructs

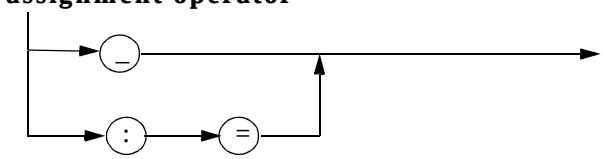
identifier



block-argument

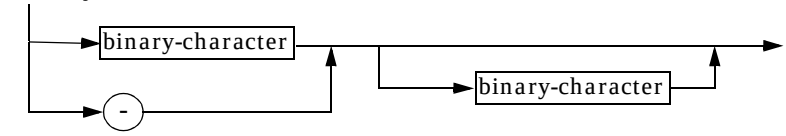


assignment-operator



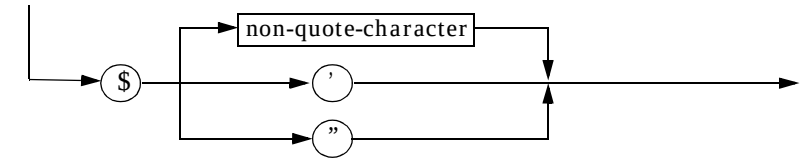
keyword

binary-selector

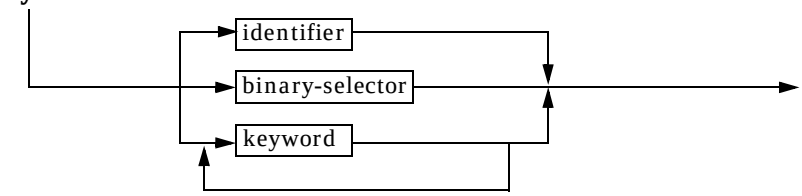


unary-selector

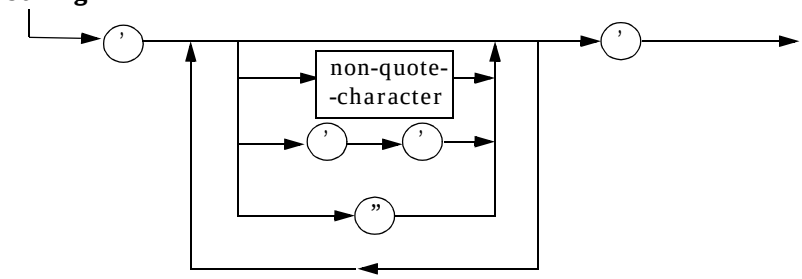
character-constant



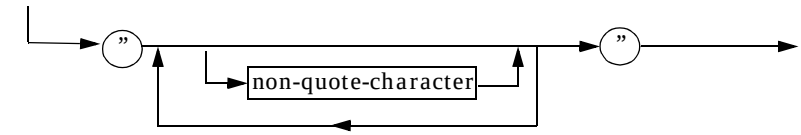
symbol



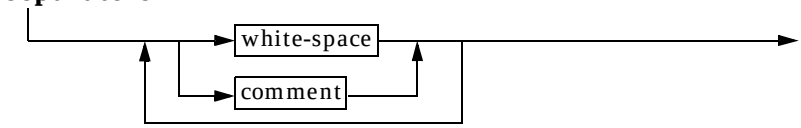
string



comment

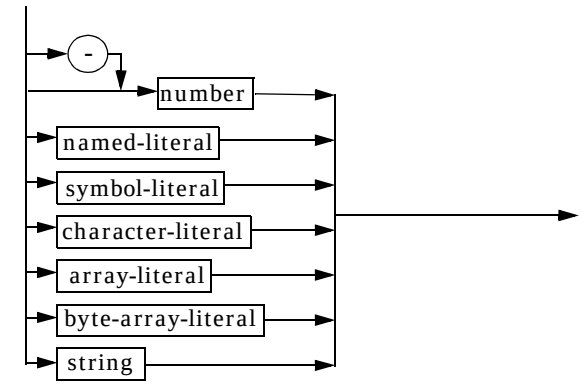


separators

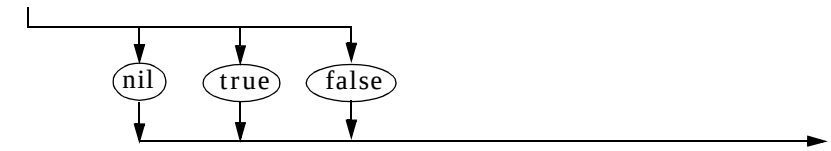


Atomic terms

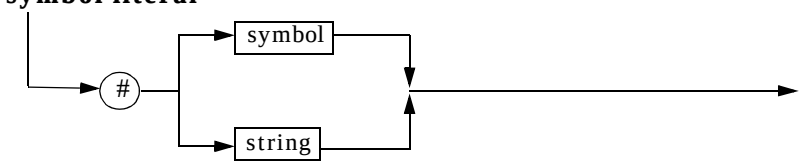
literal



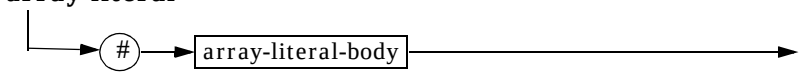
named-literal



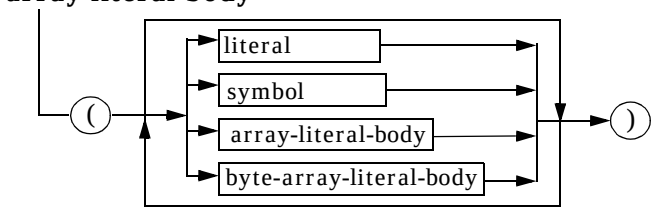
symbol-literal



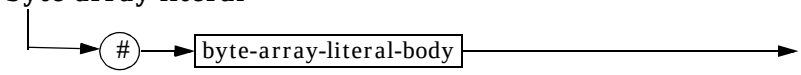
array-literal



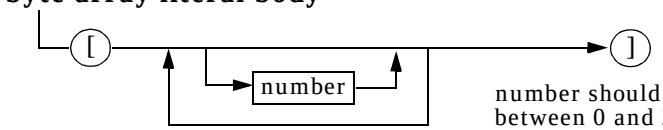
array-literal-body



byte-array-literal



byte-array-literal-body



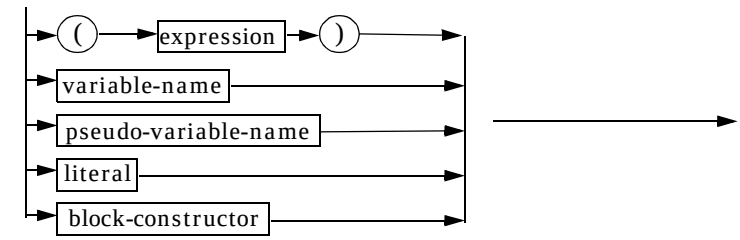
number should be an integer
between 0 and 255 inclusive

variable-name

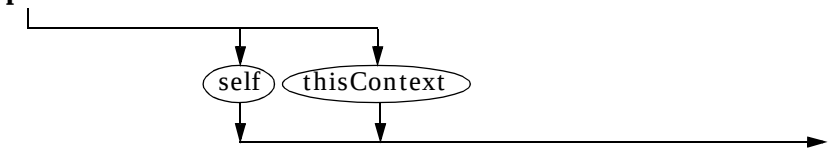
identifier should be other than
named-literal, pseudo-variable-name or super

Expressions and statements

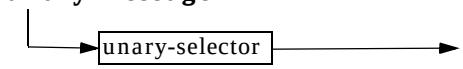
primary



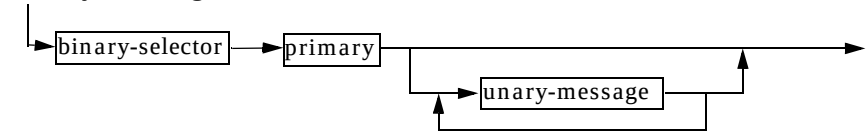
pseudo-variable-name



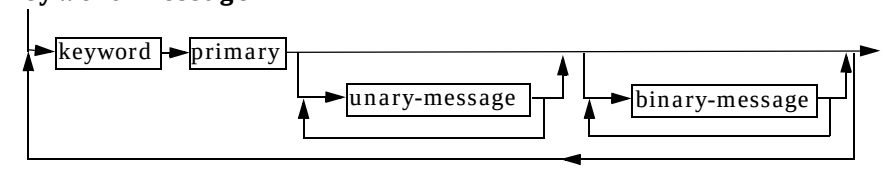
unary-message



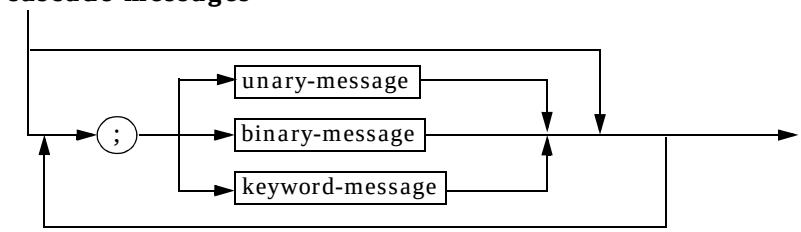
binary-message



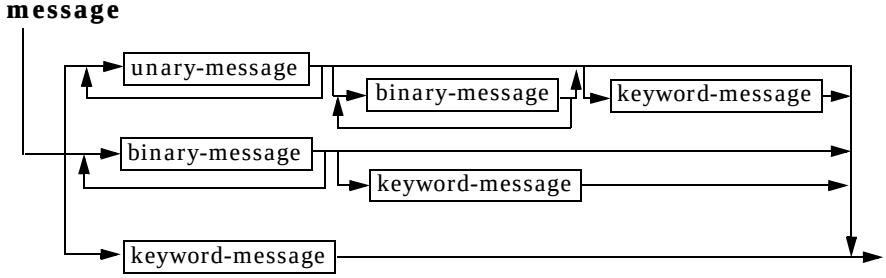
keyword-message



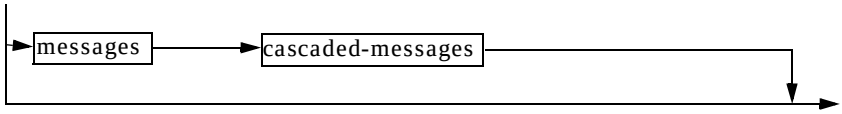
cascade-messages



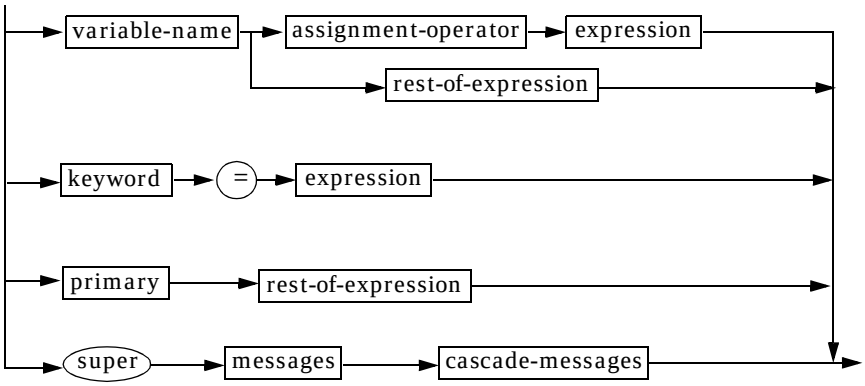
message



rest-of-expression



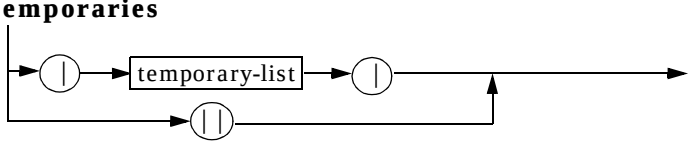
expression



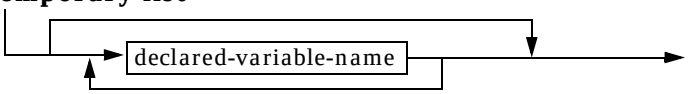
The part with “keyword = expression” should be read as “variable-name := assignment”. This is to allow a spaceless x:=3 without interpreting x: as a keyword.

expression-list

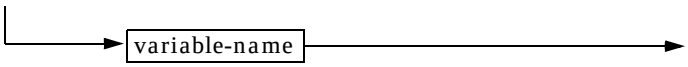
temporaries



temporary-list

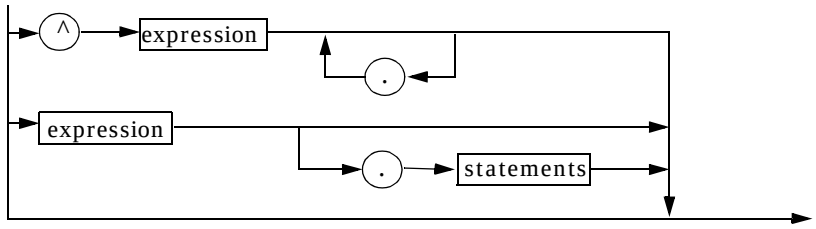


declared-variable-name

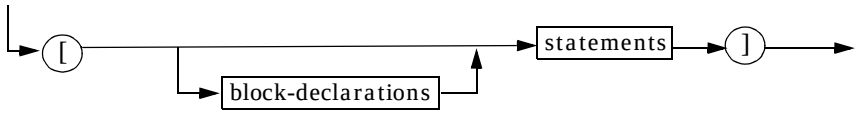


:

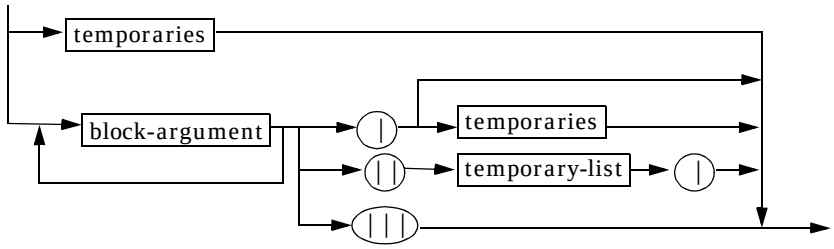
statements



block-constructor

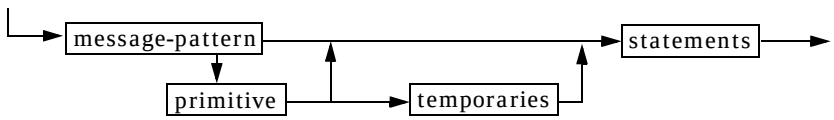


block-declarations

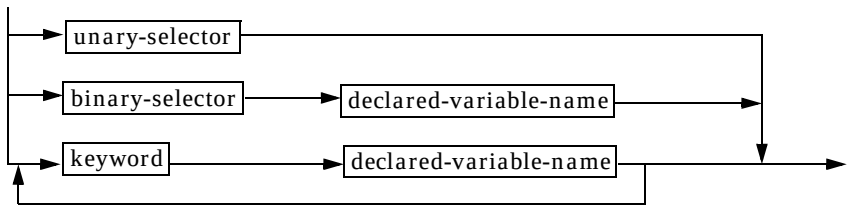


Methods

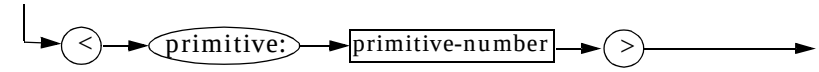
method



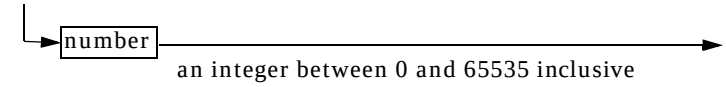
message-pattern



primitive



primitive-number



Index

Symboler

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Å

Ä

Ö

Objektorientering framstår idag som det bästa alternativet för att lösa problem i stora system. Det objektorienterade språket Smalltalk anses av många ha den bästa miljön och vara det bästa språket för att lära sig objektorienterat tänkande.

En del använder Smalltalk enbart för att lära sig objektorientering. Andra överger helt sina utvecklingsmiljöer och språk som C++ för att korta ner utvecklingstiderna och fokusera på problemlösning istället för på syntax.

Smalltalk lämpar sig för såväl prototyputveckling som storskaliga applikationer. Smalltalk är det praktiskt använda objektorienterade språk som har drivit den objektorienterade tekniken längst och behandlar allt som objekt.

Boken vänder sig till läsare som redan har viss erfarenhet av programmering. I boken ges en allmänn förklaring till termer, tekniker och praktiska exempel i objektorienterad programmering. Boken innehåller också en djup beskrivning av programspråket Smalltalk.

Denna bok kommer att följas av böcker som ingående beskriver de olika utvecklingsmiljöerna i Smalltalk och en bok om interaktionsprogrammering i Smalltalk.

Olle Bälter, född 1962 i Umeå, civ ing F.

Doktorerar på användargränssnitt vid Interaktions- och Presentations-laboratoriet vid Tekniska Högskolan i Stockholm. Flerårig erfarenhet av industriell programutveckling, bl a från Siemens. Har undervisat på KTH sedan 1984, i såväl nybörjarkurser som fortsättningskurser i programmering.

Vald av eleverna på Lantmäterisektionen till "Årets lärare" 1987. Mottog 1989 ur Hennes Majestät Drottningens hand KTH's pedagogikstipendium. Författare av två läroböcker i Pascalprogrammering.

Konsult/egen företagare på deltid. Fritiden ägnas åt beachvolleyboll, katamaransegling, fysikspex och jordenruntresor.

Per Hägglund, född 1964 i Stockholm, Datalogexamen

Doktorerar på användargränssnitt för datorstött samarbete vid Interaktions- och Presentations-laboratoriet vid Tekniska Högskolan i Stockholm. Har arbetat vid KTH sedan examen 1989 med bl a Smalltalk programmering och undervisning. i Smalltalk, datorgafik och Unix. Har aldrig blivit vald till bästa lärare ej heller sett Silvia, annat än på TV.

Försöker uppfostra tre illbattingar på ’fritiden’ och när de sover utveckla pedagogiska program för läs- och skrivsvaga.

Björn Eiderbäck, född 1959 i Stockholm, Datalogexamen.

Har arbetat vid KTH sedan 1985. Har efarenhet av objektorienterad programmering sedan 1982 och Smalltalk sedan 1986. Har flera års internationell erfarenhet av undervisning, konsultarbete och programutveckling på både högskola och industri. Undervisar för närvarande på både nybörjarkurser och fördjupningskurser i främst objektorienterad programutveckling.