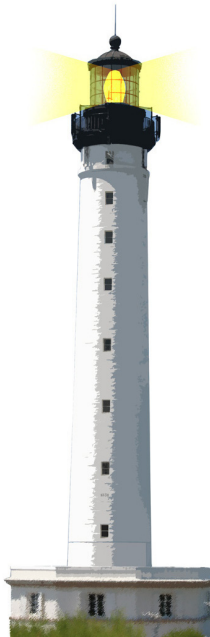
Overloading (in Java)

Why you should not use it



Goal

- What is overloading?
- Why is it a problem?
- What can you do?
- Better use real double dispatch

Overloading

Overloading: multiple methods can have the same name

```
public class OverloadingSimple {
  public String m(int a) {
    return "my parameter is an int";
  }
  public String m(String b) {
    return "my parameter is a string";
  }
  public static void main(String[] args) {
    System.out.println(new OverloadingSimple().m(123));
    System.out.println(new OverloadingSimple().m("a"));
  }
}
```

Seems nice the first two minutes

Exercise

```
class A {
  public void f(A a) { print("A.f(A)"); }
}
class B extends A {
  public void f(A a) { print("B.f(A)"); }
  public void f(B b) { print("B.f(B)"); }
}

A a = new B();
B b = new B();
a.f(a);     a.f(b);
b.f(a);     b.f(b);
```

What are the results of these 4 expressions?

Solution

```
class A {
  public void f(A a) { print("A.f(A)"); }
}
class B extends A {
  public void f(A a) { print("B.f(A)"); }
  public void f(B b) { print("B.f(B)"); }
}

A a = new B(); //dynamic type of a is B
B b = new B();
a.f(a);    a.f(b);
b.f(a);    b.f(b);
```

B.f(A)	B.f(A)
B.f(A)	B.f(B)

Explanation

Overloading + sub-typing:

- The compiler searches a **compatible signature** in the static class of the receiver and argument (use potentially subtyping) and stores it.
- During execution, the lookup algorithm searches for a method with this exact signature, **regardless of the dynamic types** of arguments.

Why a.f(b) returns B.f(A)?

```
class A { public void f(A a) { print("A.f(A)"); }}
class B extends A {
  public void f(A a) { print("B.f(A)"); }
  public void f(B b) { print("B.f(B)"); }}
```

At compile time

- Look in the static type of the receiver (here class A) and check existing method
- Use static type of the arguments since B is a subtype of A, f(A) matches
- So we will look for 'fA'

During execution

- a.f(b) looks for fA in class B because the dynamic type of a is B
- > B.f(A)

Looking at a real case: a Visitor

```
public class Shape {
  public void accept(Visitor visitor) {
    visitor.visit(this); }}
public class Visitor {
  public void visit(Shape a) {
    System.out.println("Visitor.visit(Shape)"); }}

Visitor visitor = new Visitor();
new Shape().accept(visitor); // "Visitor.visit(Shape)"
```

All is well in this perfect world...

Uh! My Visitor is not executed!

```
public class Square extends Shape {
  public void accept(Visitor visitor) {
    visitor.visit(this); }
public class SubVisitor extends Visitor {
  public void visit(Square sq) {
    System.out.println("SubVisitor.visit(Square)"); }}

Visitor visitor = new SubVisitor();
new Square().accept(visitor);  // "Visitor.visit(Shape)"
```

When method Square.accept() is compiled, the message visitor.visit(this) is bound to visit(Shape), the only compatible signature in Visitor. During execution, visit(Square) is ignored because we look for visitShape.

Possible ugly 'solutions'

One way to fix that could be to add a visit(SubA) method in Visitor (if you can change it):

```
public class Visitor {
  public void visit(Shape a) {
    System.out.println("Visitor.visit(Shape)");
  }
  public void visit(Square sq) {
    System.out.println("Visitor.visit(Square)");
  }
}
```

This works but has 3 problems:

- it breaks the modularity of visitors
- the methods accept() in Shape and Square are identical
- you can't use the visit() methods directly

In particular

The methods accept() in A and SubA are identical:

```
public class Shape {
  public void accept(Visitor visitor) {
    visitor.visit(this);
  }
}
public class Square extends Shape {
  public void accept(Visitor visitor) {
    visitor.visit(this);
  }
}
```

You could think that Square.accept() is useless because its content is the same as Shape.accept(). But it's not useless as, inside it, the message visitor.visit(this) binds to the signature visit(Square) which is what we want.

In particular

You can't use the visit() methods directly:

```
public class Visitor {
  public void visit(Shape a) {
    System.out.println("Visitor.visit(Shape)");
  }
  public void visit(Square a) {
    System.out.println("Visitor.visit(Square)");
  }
}

Visitor visitor = new Visitor();
Shape sq = new Square();
visitor.visit(sq); // "Visitor.visit(Shape)"
```

If you use the visitor directly (i.e., without going through accept() methods), the static types become important to decide which visit() method to execute.

The Real Solution

Don’t use overloading and prefer dedicated method names and double dispatch

```
public class Shape {
  public void accept(Visitor visitor) {
    visitor.visitShape(this);
  }
}

public class Square extends Shape {
  public void accept(Visitor visitor) {
    visitor.visitSquare(this);
  }
}

public class Visitor {
  public void visitShape(Shape s) {
    System.out.println("Visitor.visitShape(Shape)");
  }
  public void visitSquare(Square ss) {
    System.out.println("Visitor.visitSquare(Square)");
```

Overloaded method calls

The static type of argument is always used

- no dynamic dispatch
- force you to cast
- force you to use `getClass`

The static type of arguments is always used to resolve overloaded method calls.

- It means that your subclass definitions are **ignored**

Summary

1. overloading is nice
2. just pay attention you don’t mix it with sub-typing...
3. but OO programming is all about sub-typing...
4. don’t use overloading

A course by

S. Ducasse, G. Polito, and Pablo Tesone