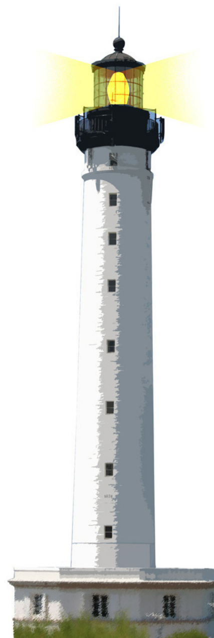
Object-Oriented Design Lecture

Overloading in Java

Why you should not use it

Stéphane Ducasse and Luc Fabresse

http://stephane.ducasse.free.fr
http://car.mines-douai.fr/luc



Pharo
http://www.pharo.org

Goal

- What is overloading?
- Why is it a problem?
- What can you do?

This lecture is in the context of the Java programming language.

- It is an incentive to understand for real double dispatch (without overloading)

Overloading

Overloading: methods sharing a 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 at first to avoid thinking about names.

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(); // take care, this is a 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(); // take care, this is a 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 is confusing in Java.
In the presence of a message send, the compiler searches a *compatible signature* and stores it. During execution, the lookup algorithm searches for a method with this exact signature, regardless of the dynamic types of arguments.

Looking at a Real Case: a Visitor

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

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

All is well in this perfect world...

Uh! My Visitor is not Executed!

```
public class SubA extends A {
  public void accept(Visitor visitor) {
    visitor.visit(this);
  }
}
public class SubVisitor extends Visitor {
  public void visit(SubA subA) {
    System.out.println("SubVisitor.visit(SubA)");
  }
}

Visitor visitor = new SubVisitor();
new SubA().accept(visitor);  // "Visitor.visit(A)"
```

When method SubA.accept() is compiled, the message visitor.visit(this) is bound to visit(A), the only compatible signature in Visitor. During execution, visit(SubA) is ignored.

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(A a) {
    System.out.println("Visitor.visit(A)");
  }
  public void visit(SubA a) {
    System.out.println("Visitor.visit(SubA)");
  }
}
```

This works but has 2 problems (see next slides):

- the methods accept() in A and SubA are identical
- you can't use the visit() methods directly

In Particular

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

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

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

In Particular

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

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

Visitor visitor = new Visitor();
A subA = new SubA();
visitor.visit(subA); // "Visitor.visit(A)"
```

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:

```
public class A {
  public void accept(Visitor visitor) {
    visitor.visitA(this);
  }
}

public class Visitor {
  public void visitA(A a) {
    System.out.println("Visitor.visitA(A)");
  }
}
```

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

Stéphane Ducasse

http://stephane.ducasse.free.fr

and

Luc Fabresse

http://car.mines-douai.fr/luc

Except where otherwise noted, this work is licensed under CC BY-NC-ND 3.0 France

https://creativecommons.org/licenses/by-nc-nd/3.0/fr/