Programming Fundamentals 2

Pierre Talbot

16 March 2021

University of Luxembourg

UNIVERSITÉ DU LUXEMBOURG

Chapter V. Subtype Polymorphism

Introductory challenge

Challenge

Add a method `ascii_art` returning the ASCII drawing of the weapon (`String` type).

Introductory challenge

Challenge

Add a method `ascii_art` returning the ASCII drawing of the weapon (`String` type).

```
class Axe { // ...
  // from http://www.chris.com/ascii/index.php?art=objects/axes
  public String ascii_art() {
    return
      " /’-./\_   \n" + // What’s wrong here?
      ":    ||,>  \n" +
      " \.-’||    \n" + // And here?
      "     ||    \n" +
      "     ||    \n" +
      "     ||    \n";
  }
}
```

Introductory challenge (text block Java 15)

Challenge

Add a method `ascii_art` returning the ASCII drawing of the weapon (`String` type).

```
class Axe { // ...
  // from http://www.chris.com/ascii/index.php?art=objects/axes
  public String ascii_art() {
    return
      """
         /’-./\\_
        :    ||,>
         \\.-’||
             ||
             ||
             ||
      """;
  }
}
```

Subtype polymorphism

Shop

Consider a weapon shop `ArrayList<Weapon> store;`, can you print the ASCII drawing of all the weapons in this store?

Subtype polymorphism

Shop

Consider a weapon shop `ArrayList<Weapon> store;`, can you print the ASCII drawing of all the weapons in this store?

Issues

- Class `Weapon` doesn't have a method `ascii_art`!
- How to view the "real or concrete type" an object of type `Weapon`? More formally, how to view its runtime type (`Axe` or `Hammer`)? Spoiler: We don't! We use overriding instead so the runtime type is automatically used.

Overriding mechanism

Override-equivalent signatures

Two method signatures are *override-equivalent* if they have exactly the same name, same parameters types and return type. Actually, the return type can be co-variant (we'll talk about that in Chapter 7).

Overriding mechanism

Override-equivalent signatures

Two method signatures are *override-equivalent* if they have exactly the same name, same parameters types and return type. Actually, the return type can be co-variant (we'll talk about that in Chapter 7).

Overriding

For all classes $T \leq Weapon$, if a method $T.m$ is *override-equivalent* to $Weapon.m$, then the method called will be the one of the smallest subclass.

Overriding mechanism

Override-equivalent signatures

Two method signatures are *override-equivalent* if they have exactly the same name, same parameters types and return type. Actually, the return type can be co-variant (we'll talk about that in Chapter 7).

Overriding

For all classes $T \leq Weapon$, if a method $T.m$ is *override-equivalent* to $Weapon.m$, then the method called will be the one of the smallest subclass.

Late-binding

Method calls are resolved at *runtime*. Indeed, we cannot guess at compile-time the runtime-type of the object. Why? Imagine the following code:

```
Weapon w;
if(a) { w = new Axe();} else { w = new Hammer(); }
w.ascii_art(); // Axe.ascii_art or Hammer.ascii_art?
```

Example overriding

```
class Weapon {
  public String ascii_art() {
    return ????;
  }
}
```

Design issue! A weapon cannot be draw in general. By the way, can a “general weapon” exist? Probably not since it is an abstract concept.

Example overriding

```
class Weapon {
  public String ascii_art() {
    return ????;
  }
}
```

Design issue! A weapon cannot be draw in general. By the way, can a "general weapon" exist? Probably not since it is an abstract concept.

Refactoring

- We must update the class `Weapon` to take into account the *new requirements*.
- Class `Weapon` must be an abstract class! An abstract class can contain attributes and methods, but some methods do not have a body.

Complete example

```java
abstract class Weapon {
  protected double damage;
  public Weapon(double damage) {
    this.damage = damage;
  }
  abstract public String ascii_art();
}

class Axe extends Weapon {
  private static final double DAMAGE = 10;
  public Axe() {
    super(DAMAGE);
  }
  public String ascii_art() {
    return
      """
       <|>
        |
        |
      """;
  }
}
```

Complete example (next)

```
class Hammer extends Weapon {
  private static final double DAMAGE = 20;
  public Hammer() {
    super(DAMAGE);
  }
  public String ascii_art() {
    return
      """
       _ _
      |_|_|
        |
        |
      """;
  }
}

public class TestWeapon {
  public static void main(String[] args) {
    ArrayList<Weapon> store = new ArrayList<>();
    store.add(new Hammer());
    store.add(new Axe());
    for(Weapon w : store) {
      System.out.println(w.ascii_art());
    }
  }
}
```

What to remember about subtype polymorphism?

- “Polymorphism” because a type can have several forms (the subtypes, *i.e.*, in Java the subclasses).
- *Overriding mechanism* allowing to redefine a behavior more precisely.
- Methods are selected at runtime (*late-binding*).
- At compile-time, the methods are selected according to the rules of *ad-hoc polymorphism* and *overloading*.

Polymorphism Cocktail

Mixing overloading and overriding

- We can mix ad-hoc polymorphism and subtype polymorphism together.
- We first select the method via *overloading* (selected at compile-time).
- Then, at runtime, we check if *overriding* can apply (the signature must be *override-equivalent* to the one selected at compile-time).

Exercise

```
class A {
  void m(A x, B y){System.out.println ("1");}
  void m(B x, A y){System.out.println ("2");}
}
class B extends A {
  void m(B x, B y){System.out.println ("3");}
}
class C extends B {
  void m(B x, B y){System.out.println ("4");}
  void m(C x, C y){System.out.println ("5");}
  void m(B x, A y){System.out.println ("6");}
}
```

Exercise (part 2)

For each call, what is the method selected at compile-time, and then at runtime?

```
class PolymorphicCocktail {
  public static void main(String[] args) {
    A a1 = new A();
    B b1 = new B();
    C c1 = new C();
    A a2 = b1;
    A a3 = c1;
    B b2 = c1;

    a1.m(b1,c1);
    b1.m(b1,c1);
    c1.m(b1,c1);
    a1.m(a1,a1);

    a2.m(b1,c1);
    a3.m(b1,c1);
    b2.m(b1,c1);
    // ... (more in the next slide)
```

Exercise (part 3)

```
    A a1 = new A();
    B b1 = new B();
    C c1 = new C();
    A a2 = b1;
    A a3 = c1;
    B b2 = c1;
    // ...

    a1.m(b2,a3);
    a2.m(b2,a3);
    a3.m(b2,a3);

    a1.m(c1,b1);
    b1.m(c1,b1);
    b2.m(c1,b1);
    c1.m(c1,b1);
  }
}
```

Correction

```
class PolymorphicCocktail {
  public static void main(String[] args) {
    A a1 = new A();
    B b1 = new B();
    C c1 = new C();
    A a2 = b1;
    A a3 = c1;
    B b2 = c1;

    // solution of the form ‘(compile-time) / (execution-time)’
    a1.m(b1,c1); // ambiguous between (1) and (2)
    b1.m(b1,c1); // (3)/(3)
    c1.m(b1,c1); // (4)/(4)
    a1.m(a1,a1); // no suitable method found

    a2.m(b1,c1); // ambiguous between (1) and (2)
    a3.m(b1,c1); // ambiguous between (1) and (2)
    b2.m(b1,c1); // (3)/(4)

    a1.m(b2,a3); // (2)/(2)
    a2.m(b2,a3); // (2)/(2)
    a3.m(b2,a3); // (2)/(6)
    // ... (more in the next slide).
```

Correction (part 2)

```
    A a1 = new A();
    B b1 = new B();
    C c1 = new C();
    A a2 = b1;
    A a3 = c1;
    B b2 = c1;

    a1.m(c1,b1); // ambiguous between (1) and (2)
    b1.m(c1,b1); // (3)/(3)
    b2.m(c1,b1); // (3)/(4)
    c1.m(c1,b1); // (4)/(4)
  }
}
```

Complementary resources

The Java Language Specification

- Link: `http://docs.oracle.com/javase/specs/` (Java 15):
- §8.4.8: *overriding*.
- §8.4.9: *overloading*.
- §15.12: Method invocation (detailed steps performed by the compiler).
- Hard to read and understand because it is exhaustive!
- Nonetheless the best resource to find precise explanations.