# Programming Fundamentals 2

Pierre Talbot

11 March 2021

University of Luxembourg

# Chapter IV. Ad-hoc Polymorphism

# Polymorphism

- Fundamental concept in computer science.
- It means that "something can exist in different forms".
- A same type can have different behaviors.

# Polymorphism

- Fundamental concept in computer science.
- It means that “something can exist in different forms”.
- A same type can have different behaviors.

## Different kind of polymorphisms

- Ad-hoc polymorphism.
- Subtyping polymorphisme (through inheritance).
- Casting polymorphisme.
- Parametric polymorphism (through generics).

# Compile-time and runtime types

# Compile-time type

## Compile-time type

- The type is associated to the variable during the compilation.
- It is the type written when declaring a variable, *e.g.*, `Integer i`.
- Variables with primitive types can only have compile-time type.

# Exercise: compile-time type

```
class WeaponStore{
  Weapon cheater = new Weapon(100);
  Weapon axe = new Axe();
  Weapon hammer = new Hammer();
  int number_weapons = 3;
  Number extra_damage = new Integer(42);

  public int price(Weapon w) { /* ... */ }
}
//... In main function.
WeaponStore store = new WeaponStore();
store.price(new Axe());
store.price(new Weapon(22));
```

# Solution: compile-time type

```
class WeaponStore{
  Weapon cheater = new Weapon(100); // Weapon
  Weapon axe = new Axe(); // Weapon
  Weapon hammer = new Hammer(); // Weapon
  int number_weapons = 3; // int
  Number extra_damage = new Integer(42); // Number

  // The compile-time type of w is Weapon
  public int price(Weapon w) { /* ... */ }
}
//... In main function.
WeaponStore store = new WeaponStore(); // WeaponStore
store.price(new Axe()); // the temporary variable has type Axe.
store.price(new Weapon(22)); // temporary has type Weapon.
```

# Runtime type

## Runtime type

- The “real type” of the variable, as initialized at runtime.
- The runtime type ($c_1$) is always a subclass or identical ($c_1 \leq c_2$) to the compile-time type ($c_2$).
- For instance, `Axe axe = new Weapon(39);` does not make sense. A weapon *is not* an axe, a weapon can be many other things.
- Moreover, technically, how would we initialize the remaining members of `Axe`?

# Example: runtime types

```java
class WeaponStore{
  Weapon cheater = new Weapon(100);
  Weapon axe = new Axe();
  Weapon hammer = new Hammer();
  int number_weapons = 3;
  Number extra_damage = new Integer(42);

  public int price(Weapon w) { /* ... */ }
}
//... In main function.
WeaponStore store = new WeaponStore();
store.price(new Axe());
store.price(new Weapon(22));
```

# Solution: runtime types

```
class WeaponStore{
  Weapon cheater = new Weapon(100); // Weapon
  Weapon axe = new Axe(); // Axe
  Weapon hammer = new Hammer(); // Hammer
  int number_weapons = 3; // int
  Number extra_damage = new Integer(42); // Integer

  // The dynamic type of w can be
  // Weapon, Axe or Hammer.
  public int price(Weapon w) { /* ... */ }
}
//... In main function.
WeaponStore store = new WeaponStore(); // WeaponStore
store.price(new Axe()); // the temporary variable has type Axe.
store.price(new Weapon(22)); // temporary has type Weapon.
```

# Ad-hoc Polymorphism

# Ad-hoc polymorphism (overloading)

## Introductory challenge

- Create a class `Monster` and `Obstacle` each having a health points attribute and a method to decrease these health points.
- Add two methods to `Axe` and `Hammer` to attack the monsters and obstacles.
- The dammage of the axe on monsters is weigthed by 0.8, and on obstacles by 1.2.
- For the hammer, we have 1.4 and 0.7.

## Thoughts on method names

Did you call the method to decrease the health points `set_life` or similarly?

# Thoughts on method names

Did you call the method to decrease the health points `set_life` or similarly?

## Coding style

- Methods such as `set_*` and `get_*` are *bad names* because they lead to imperative-style code, and not the “service-oriented” approach of OO.
- They somewhat break encapsulation because they expose internal attributes.
- A method should give a *service*, it must show in the name.
- It’s hard to find good names, but very important.
- Sometimes, we want to have records (and not objects), in which case you can use immutable records or PODS and POJO, `http://en.wikipedia.org/wiki/Plain_old_data_structure`).

# First solution

```java
class Monster {
  private double life = 100;
  public void hit_me(double damage) { life = Math.max(0, life - damage); }
}
class Obstacle { /* similar */ }

class Axe {
  static final double MONSTER_DAMAGE_RATIO = 0.8;
  static final double OBSTACLE_DAMAGE_RATIO = 1.2;

  public void attack_monster(Monster m) {
    m.hit_me(damage * MONSTER_DAMAGE_RATIO);
  }

  public void attack_obstacle(Obstacle o) {
    o.hit_me(damage * OBSTACLE_DAMAGE_RATIO);
  }
}
class Hammer { /* similar */ }
```

# Observation

```
public void attack_monster(Monster m)
```

Anything wrong with this method?

# Observation

```
public void attack_monster(Monster m)
```

Anything wrong with this method?

### Coding style

You should avoid any repetition, in the code, but also in the names. This method signature already indicates we attack a monster, no need to repeat it.

# Second solution

```
class Monster {
  private double life = 100;
  public void hit_me(double damage) { life = Math.max(0, life - damage); }
}
class Obstacle { /* similar */ }

class Axe {
  static final double MONSTER_DAMAGE_RATIO = 0.8;
  static final double OBSTACLE_DAMAGE_RATIO = 1.2;

  public void attack(Monster m) {
    m.hit_me(damage * MONSTER_DAMAGE_RATIO);
  }

  public void attack(Obstacle o) {
    o.hit_me(damage * OBSTACLE_DAMAGE_RATIO);
  }
}
class Hammer { /* similar (constants change) */ }
```

# Overloading

### Definition

Overloading is a *compile-time mechanism* allowing us to use a same name for multiple methods, when those have a similar role.

# Overloading

### Definition

Overloading is a *compile-time mechanism* allowing us to use a same name for multiple methods, when those have a similar role.

### Compile-time

It is only based on the compile-time type, the runtime type plays no role, and the method calls are resolved at compile-time (aka. *static binding*).

# Overloading

When calling `obj.method(`$a_1$`, ..., `$a_n$`)`, how to be sure of which methods will be selected at compile-time? (trivial steps in grey).

1. Identify the classes to explore (compile-time type of `obj` + super classes).
2. Locate the *accessible* methods (public or protected in super classes) with the same name.
3. Select the methods with the same arity (numbers of arguments).
4. Select the *applicable* methods, *i.e.*, those with types of $a_i$ are $\leq T_i$, $T_i$ being the type of the parameter.
5. Apply an algorithm to select *the most specific method*.

Note: The return type does not matter.

# Overloading resolution algorithm

This algorithm can be different depending on the language. Even between different versions of a same language (Java 1.2 vs Java 1.5 or later). Here, we present the most recent for Java.

1. Let $A_i$ be the types of arguments, and $P_i$ the types of the parameters.
2. For each argument, compute the "inheritance distance" between $A_i$ and $P_i$, if $A_i \equiv P_i$ then the distance is 1.
3. Add distances.
4. The method with the smallest distance is selected.
5. If several distances are identical, then a compile-time error *ambiguous call* occurs.

# Notes on overloading

- It is usually used when methods are non-ambiguous:
  - A different arity.
  - The parameters are not connected through inheritance.
- Otherwise, the programmer must manually execute the resolution algorithm to be sure of which method is called.
- Therefore, you should use it carefully and keep it simple.
- Generally, the philosophy adopted by the Java librairies.

# Exercise I

### Don’t repeat yourself

Use a parent class `Destructible` extracting the common code in `Monster` and `Obstacle`.

# Exercise I

## Don’t repeat yourself

Use a parent class `Destructible` extracting the common code in `Monster` and `Obstacle`.

## Solution

```
class Destructible {
  protected double life = 100;
  public void hit_me(double damage) { life = Math.max(0, life - damage); }
}
class Monster extends Destructible { /* ... */}
class Obstacle extends Destructible { /* ... */ }
```

# Exercise II

What is the method called, or the error, if for each object *o* declared below, we write `axe.attack(o)`?

```
class Axe {
  public void attack(Monster m) {} // (1)
  public void attack(Obstacle o) {} // (2)
  public void attack(Destructible d) {} // (3)
}
```

```
Destructible dmonster = new Monster();
Destructible dobstacle = new Obstacle();
Monster monster = new Monster();
Obstacle obstacle = new Obstacle();
```

# Solution: Exercise II

```
Destructible dmonster = new Monster();
Destructible dobstacle = new Obstacle();
Monster monster = new Monster();
Obstacle obstacle = new Obstacle();

axe.attack(dmonster); // (3)
axe.attack(dobstacle); // (3)
axe.attack(monster); // (1)
axe.attack(obstacle); // (2)
```

### Compile-time

Don't forget that *overloading* only looks at the compile-time type!

# Exercice III

What about these examples?

```
class Axe {
  public void attack(Monster m, Obstacle o) {} // (1)
  public void attack(Destructible d, Monster m) {} // (2)
  public void attack(Monster m, Destructible d) {} // (3)
}

Destructible dmonster = new Monster();
Destructible dobstacle = new Obstacle();
Monster monster = new Monster();
Obstacle obstacle = new Obstacle();

axe.attack(monster, obstacle);
axe.attack(dobstacle, monster);
axe.attack(dobstacle, dmonster);
axe.attack(dmonster, dmonster);
axe.attack(monster, monster);
axe.attack(monster, dobstacle);
```

# Solution: Exercise III

```
class Axe {
  public void attack(Monster m, Obstacle o) {} // (1)
  public void attack(Destructible d, Monster m) {} // (2)
  public void attack(Monster m, Destructible d) {} // (3)
}

Destructible dmonster = new Monster();
Destructible dobstacle = new Obstacle();
Monster monster = new Monster();
Obstacle obstacle = new Obstacle();

axe.attack(monster, obstacle); // (1)
axe.attack(dobstacle, monster); // (2)
axe.attack(dobstacle, dmonster); // error: no such method
axe.attack(dmonster, dmonster); // error: no such method
axe.attack(monster, monster); // error: ambiguous call between
                              // (2) and (3)
axe.attack(monster, dobstacle); // (3)
```

# What to remember of ad-hoc polymorphism?

- Called polymorphism because a method can have several forms (all the methods with an identical name).
- *Overloading* mechanism allowing us to use a same name for different implementations. However, these methods should be connected semantically.
- The method called is chosen at compile-time (*static-binding*).