Lecture 9
Subclassing, Inheritance and Polymorphism

Housekeeping

- Exams are ready for collection from Studentexpeditionen in EDIT building, 4th floor
- Exam review period is Wednesday 29 January 12.00-13.00 in EDIT 6110
- Threshold for passing grade (3) lowered to 13 points
- Thresholds for other grades (4, 5) unchanged

IDE

- An IDE (Integrated Development Environment) provides:
  - a source code editor
  - code completion
  - highlighting of many (not all!) errors
  - automatic build tools
  - debugger
  - unit testing
  - …
- IntelliJ IDEA is installed on the university machines

OOP Recap

- A *class* is a recipe for making objects.
- Classes can have *constructors*, *fields* and *methods*.
- You can create any number of *objects* from the same class.

OOP Recap

- Each object has its own copy of the (non-static) fields and methods defined in its class.
  - You need an object to use them:

```
Pony pony = new Pony("Fluttershy");
pony.greet("everyone");
```

- Static methods and fields are shared between all objects.
  - You can access them using the name of their class:

```
System.out.println(Pony.MAX_AGE);
```

Using Objects

Code Example: Text-Based Fighting Game

Overloading

```
public Circle(double r, String color) {
   ...
}

public Circle(double r) {
   ...
}
```

Overloading

```
public class Enemy {
   public void kick(int damage) {
      ...
   }
   public void kick(Player kicker) {
      ...
   }
}

...

Player john = new Player("John McClane");
Enemy hans = new Enemy("Hans Gruber");
hans.kick(5);
hans.kick(john);
```

Overloading

Code Example: Better Kicks

Subclassing

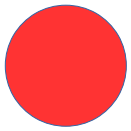
We can make a class A a *subclass* of a class B.

This means: every object of type A is also an object of type B.

(Think ‘subset’ in mathematics.)

Other ways of saying the same thing:

- B is a *superclass* of A
- A *inherits* from B
- A is *derived* from B



Subclassing

To make A a subclass of B:

```
class A extends B {

  …

}
```

All the public members of B are now members of A.

Subclassing

- The subclass relation is *transitive*.
  - If `A` is a subclass of `B`, and `B` is a sublass of `C`, then `A` is also a subclass of `C`
  - `Samon` is a `Fish`, `Fish` is a `Vertebrate`, and `Vertebrate` is an `Animal` – thus `Salmon` is an `Animal`
- `Object` is a superclass of *all other classes*!
- This is why all objects have `equals`, `toString`, etc.: they are inherited from `Object`

Subclassing

Code Example: Subclassing Fighters

Overriding

```
class B {
  public int f(int x) {…}
}
class A extends B {
 …
}
```

The class A has a method f. Calling it executes the same code is calling it in B.

We can also write a new version of f in A. This is called *overriding* f.

```
class A extends B {
 @Override
 public int f(int x) {...}
}
```

The `@Override` annotation is not required, but helps with finding bugs and readability.

Overriding vs. Overloading

- Overloading: decided at **compile time**

```
public class Enemy extends Fighter { … }

public void punch(Enemy e) {
  System.out.println("punched an enemy");
}
public void punch(Fighter e) {
  System.out.println("punched a fighter");
}

Fighter someone = new Enemy(…);
punch(someone);
```

- Output: `punched a fighter`

Overriding vs. Overloading

- Overriding: decided at **run time**

```
public class Fighter {
  public void punch() {
    System.out.println("fighter got punched");
  }
}

public class Enemy extends Fighter  {
  @Override
  public void punch() {
    System.out.println("enemy got punched");
  }
}

Fighter someone = new Enemy(…);
someone.punch();
```

- Output: `enemy got punched`

Overriding vs. Overloading

- Overriding: decided at **run time**

```
public class Fighter {
  public void punch() {
    System.out.println("fighter got punched");
  }

}

Fighter someone = new Enemy(…);
someone.punch();
```

Object:
run time

Argument list:
compile time

- Output: `enemy got punched`

Access Modifiers

A member of a class A may be:

- `public` – can be accessed anywhere
- `private` – can only be accessed from within A
- `protected` – can be accessed from within A and the subclasses of A, and other classes in the same package (see later weeks)
- package-private – see later weeks

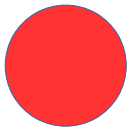
The `super` keyword

Let A be a subclass of B.

Inside the class A, the keyword `super` has two uses:

- It refers to the current object as if it were an object of class B, letting you use the methods and fields of class B.
- **As the first line of a constructor,** it invokes a constructor of B.

(Compare with the keyword `this`.)



A Note About Constructors

Let A be a subclass of B.

- The **first thing** that a constructor of A does is call a constructor of B.
- If you do not use the `super` keyword, then Java automatically calls the no-argument constructor of B.
- If you do not use the `super` keyword and B does not have a no-argument constructor, the code will not compile.

Object-Oriented Principles

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

These principles had been recognised as useful ways to organise programs for a long time.

When we created *object-oriented languages*, we represented these principles in the syntax of the language.

Abstraction

Also known as *information hiding.*

The outside world should only see relevant information. Details should be hidden.

The outside world “sees” the public methods – their signatures (names and arguments). It does not know how they are implemented. It does not see private methods and private fields.

Encapsulation

Data and the code that uses that data should be found together.

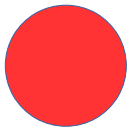
Change in one part of a program should not require change in another part.

Closely related to abstraction.

Inheritance

Each subclass should define only the fields and methods that are unique to it.

Shared behaviour should be inherited from the parent class.



Single Inheritance

- A class can only inherit from *one* class in Java (no multiple inheritance)
  - A class *can* implement multiple interfaces (next week)

Polymorphism

Call a method on objects of different classes should do different things.

I call the same method on objects of different classes, and trust that each object will do the right thing.

Closely related to inheritance.

Object Oriented Principles

- **Abstraction** and **Encapsulation** are represented by the concept of a **class**
- **Inheritance** is represented by the concept of **subclassing**
- **Polymorphism** is represented by the concept of **overriding** methods (and overloading methods)

Advantages

- Easier to *maintain* and *modify* code
  - Small change to one object instead of searching through the whole codebase
  - If the code is well designed!
  - “Code will always be changed” - bug fixes, new features
- Code reuse(?)
- Unit testing

Reading and Exercises

- Reading
  - 2.10
  - Chapter 4
  - 10.1-10.6, 10.12
- Exercises
  - 4.6 (all), 10.14 (exercises 1-6)
  - The Object-Oriented Zoo (see course website)
- Labs
  - Get started on lab 3