Object-Oriented Analysis & Java

Lecture 3

Florentin Bekier

Inheritance

Reusing code

- One of the main concern of a developer
- Don’t rewrite an existing code (use APIs or frameworks)

Class inheritance

- A class can inherit from another class by using the keyword `extends`
- Terminology:
  - Superclass (parent): the class being inherited from
  - Subclass (child): the class that inherits from another class
- A subclass inherits of all methods and attributes from the superclass
- A subclass is a specialization of the superclass
  - An instance of a subclass is thus also of the type of the superclass

Class inheritance

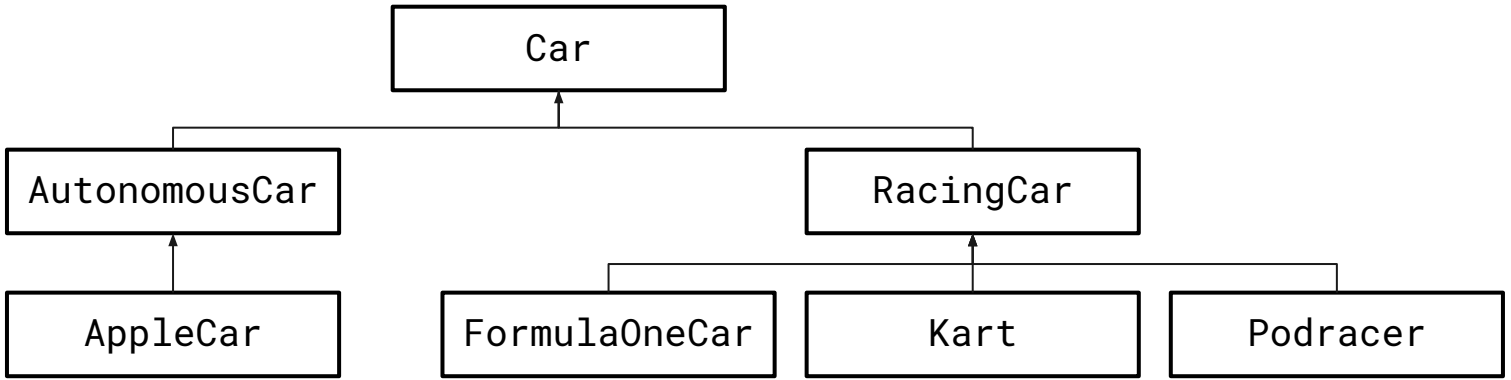
```
public class RacingCar extends Car {

    ...

}
```

Class inheritance

- No multiple inheritance in Java, a class can only inherits from one class
- Inheritance links allow to define a hierarchy between classes:



Class `Object`

- In Java, a class inherits of the class `Object` by default
  - Every object can use the methods defined by the class `Object`, such as `toString()`, `equals(Object)`, `clone()`, `hashCode()` or `getClass()`

Factoring/extending state and behaviour

- Methods and attributes (accessible) of the parent are available in their children
- A child class can provide their own methods and attribute
- A child can redefine an existing method from its parent
  (= specialization)

Keyword super

- When redefining a method inside a subclass, you need to use this keyword to make reference to the inherited method

Keyword super

```
public class Car {
    public String toString() {
        return "Car";
    }
}

public class AutonomousCar extends Car {
    public String toString() {
        return "Autonomous " + super.toString();
    }
}
```

Casting objects

- A reference has only one type, an object can have several (polymorphism)
- Casting: from a reference to an object, create another from another type, to the same object
- Upcast: change to a less specific class
- Downcast: change to a more specific class
    - **Be careful** when downcasting!

Abstract classes

Abstraction

- In Java, abstraction can be achieved with either **abstract** classes or **interfaces**
- The `abstract` keyword is a non-access modifier, used for classes and methods
  - Abstract class: a restricted class that cannot be used to create objects (it must be inherited from another class)
  - Abstract method: a method of an abstract class that has no body

Declaring an abstract class

- An abstract class can have both abstract and regular methods:

```
public abstract class Vehicle {
    public abstract void start();
    public void stop() {
        System.out.println("Vehicle stopped");
    }
}
```

Interfaces

Interface

- In Java, an interface is a **completely abstract class**
- You can use it to to group related methods with empty bodies
- A class that implements an interface must override all of its methods
- An interface cannot contain a constructor and cannot be used to create objects

Declaring an interface

- Interfaces are declared as classes, but using the keyword `interface`
- No access modifier on methods: always public

```
public interface Reparable {
    void repair();
}
```

Implementing interfaces

- Classes can implement interfaces using the keyword `implements`
- When a class implements an interface, it must declare the methods as public

```
public class Car implements Reparable {
    @Override
    public void repair() {
        // ...
    }
}
```

Interfaces and classes

- A class can implement multiple interfaces:
  - `class Car implements Reparable, Vehicle { ... }`
  - No multiple inheritance in Java: need to use interfaces
- Variables can be of an interface type, just like they can be of a class type

Extending interfaces

- An interface can be extended using the keyword `extends`

```
public interface Spacecraft extends Vehicle {
    void travelThroughHyperspace();
}
```

Enumerated types

Declaring & accessing enums

```
public enum Fuel {
    GASOLINE, DIESEL, LPG;
}
```

- An enumerated type is a class representing a defined set of values
  - The values are **static** and **final**
- To declare an enumerated type, use the `enum` keyword
- You can access an enum like this: `Fuel f = Fuel.GASOLINE;`
- You can have an enum inside another class

Enums and switch

Enums are often used in switch statements to check for corresponding values:

```
switch (f) {
     case GASOLINE :
          // ...
     case DIESEL:
          // ...
     case LPG:
          // ...
     default:
          // ...
}
```

Loop through an enum

- The static method `values()` of an enum returns an array of all enum constants
- You can use it to loop through the constants of an enum

```
for (Fuel fuel : Fuel.values()) {
    System.out.println(fuel);
}
```

Other methods

- The method `name()` returns the name of an enum constant as a string (ex: “GASOLINE”) and the method `ordinal()` returns its index
- The static method `valueOf(String s)` returns, if it exists, the instance whose reference corresponds to the string s

Assign values to the constants

```
public enum Fuel {
    GASOLINE(1.39), DIESEL(1.48), LPG(0.91);

    private final double price;

    private Fuel(double price) {
        this.price = price;
    }

    public double getPrice() {
        return this.price;
    }
}
```