2025 — 2026

1TI – 1ACS



Back-End Development

Java Inheritance, Abstract Classes en Interfaces

N. Jacobs, J. Pieck, E. Steegmans, B. Van Impe, S. Van Peborgh, V. Witters

CONTENTS

- What is inheritance
- Polymorphism
- Abstract classes
- Interfaces

What is inheritance
Polymorphism
Abstract classes
Interfaces

What is inheritance

- Inheritance
- Inheritance of the class *Object*
- Final

WHAT IS INHERITANCE

- A class is a container that groups data and actions
  - Data is stored using variables
  - Actions are defined using methods

- By programming and adding classes yourself, you increase the possibilities of what Java can do
  - You are adding your own building blocks for more complex programs

- A class can build on another class via *inheritance*

WHAT IS INHERITANCE

```
public class Part {

    private String identifier;
    private String producer;
    private String description;

    public Part(String identifier, String producer,
        String description) {
        this.identifier = identifier;
        this.producer = producer;
        this.description = description;
    }

    public String getIdentifier() {
        return identifier;
    }

    public String getProducer() {
        return producer;
    }

    public String getDescription() {
        return description;
    }
}
```

```
public class Engine {

    private String identifier;
    private String producer;
    private String description;
    private int horsepower;

    public Engine(String identifier, String producer,
        String description, int horsepower) {
        this.identifier = identifier;
        this.producer = producer;
        this.description = description;
        this.horsepower = horsepower;
    }

    public String getIdentifier() {
        return identifier;
    }

    public String getProducer() {
        return producer;
    }

    public String getDescription() {
        return description;
    }

    public int getHorsepower() {
        return horsepower;
    }
}
```

WHAT IS INHERITANCE

```
public class Engine extends Part {

    private int horsepower;

    public Engine(String identifier, String producer,
    String description, int horsepower) {
        super(identifier, producer, description);
        this.horsepower = horsepower;
    }

    public int getHorsepower() {
        return horsepower;
    }
}
```

```
public static void main(String[] args) {
    Engine engine = new Engine("A102", "Motor Company",
            "A small motor", 2);

    System.out.println(engine.getDescription());
    System.out.println(engine.getHorsepower());
}
```

- Via *extends*, a class can build on top of another class
- All fields and methods are also accessible to the inheriting class
- Used to avoid duplication in similar classes

WHAT IS INHERITANCE

```
public class Engine extends Part {

    private int horsepower;

    public Engine(String identifier, String producer, String description, int horsepower) {
        super(identifier, producer, description);
        this.horsepower = horsepower;
    }

    public String printAllInfo() {
        return super.getDescription() + super.getProducer() + super.getIdentifier() + horsepower;
    }
}
```

- Via *super*, the constructor of the *superclass* is called
  - Must be the first statement in a constructor
- You can also use methods and fields of the superclass with super
  - If they have the right visibility

WHAT IS INHERITANCE — VISIBILITY

```
public class Engine extends Part {

    // ...

    public String printAllInfo() {
        return super.getDescription() + super.getProducer()
            + super.getIdentifier() + horsepower;
    }
}
```

```
public class Part {

    private String identifier;
    private String producer;
    private String description;

    public Part(String identifier, String producer,
        String description) {
        this.identifier = identifier;
        this.producer = producer;
        this.description = description;
    }

    public String getIdentifier() {
        return identifier;
    }

    protected String getProducer() {
        return producer;
    }

    protected String getDescription() {
        return description;
    }
}
```

- *protected* fields and methods are available in the superclass and in all inheriting classes (*subclasses*)
- Now you know public, protected and private

WHAT IS INHERITANCE — INHERITANCE

- You can keep inheriting, each level always has access to what the levels above it have made available
  - For example: Animal > Vertebrate > Mammal > Giraffe
- Each class implicitly extends the class Object
  - This is how a class gets the standard toString() method that prints out the memory address
  - For this reason, instances of a class are also called objects
- A class can inherit from a maximum of 1 other class

WHAT IS INHERITANCE — FINAL

```
public final class Giraffe {

    // ...
}
```

```
public class SouthernGiraffe extends Giraffe {} // Error!
```

- By adding the keyword *final* to a class definition, you indicate that no inheritance is allowed from this class
- Often used for built-in classes (e.g. String)

WHAT IS INHERITANCE — WHEN TO USE

- Inheritance is used to model an "is a" relationship
- A Giraffe *is an* Animal
  - It is plausible that a Giraffe has the same characteristics as an Animal

- Modelling relationships other than an "is a" relationship via inheritance leads to non-intuitive and fragile codebases

EXERCISE @CLASS

- exercise1



What is inheritance

Polymorphism

Abstract classes

Interfaces

Polymorphism

POLYMORPHISM

- An object can be multiple types
- Remember (Object) > Animal > Vertebrate > Mammal > Giraffe
  - An object of the class Giraffe is of the types Object, Animal, Vertebrate, Mammal and Giraffe
- This property is called *polymorphism*
  - Literally translated, an object is "many forms" at the same time

POLYMORPHISM

```
public static void main(String[] args) {
    Giraffe tomTheGiraffe = new Giraffe(); // Ok, a giraffe is a giraffe
    Animal marieTheGiraffe = new Giraffe(); // Ok, a giraffe is an animal
    Verbetrate janTheMammal = new Mammal(); // Ok, a mammal is a vertebrate
    Giraffe marieTheAnimal = new Animal(); // Not ok, an animal is not necessarily a giraffe

    Animal tomTheAnimal = tomTheGiraffe; // Ok, a giraffe is an animal
    Giraffe janTheGiraffe = janTheMammal; // Not ok, a mammal is not necessarily a giraffe
    Giraffe marieTheGiraffe2 = (Giraffe) marieTheGiraffe; // Ok, the underlying type of the variable is a Giraffe
}
```

- An object can be multiple types and fit into a variable of each of its types
- You can cast downwards and upwards

POLYMORPHISM

```
public static void main(String[] args) {
    Giraffe tom = new Giraffe(); // All methods from Giraffe are available to call on variable tom
    Animal marie = new Giraffe(); // All methods from Animal are available to call on variable marie
    Verbetrate jan = new Mammal(); // All methods from Vertebrate are available to call on variable jan
 }
```

- Available methods to call depend on the type of the variable
  - Java is statically typed

POLYMORPHISM – OVERRIDING

```
public static void main(String[] args) {
    Animal marieTheGiraffe = new Giraffe();
    Animal janTheMammal = new Mammal();

    System.out.println(marieTheGiraffe.whoAreYou()); // Giraffe
    System.out.println(janTheMammal.whoAreYou()); // Mammal
}
```

- What if different levels define the **same method**?
  - The method used depends on **the type of the object** at runtime
    - The type of the object is determined by the constructor that was called
  - Remember that a method is defined by its name and its arguments
- We say that a class *overrides* a method of the superclass

EXERCISE @CLASS

- exercise2



What is inheritance

Polymorphism

Abstract classes

Interfaces

Abstract Classes

ABSTRACT CLASSES

```
public abstract class Animal {

    private int age;

    // ...

    public String whoAreYou() {
        return "";
    }
}
```

- Adding the keyword *abstract* makes a class abstract
  - You cannot create instances of an abstract class
  - For all other things, an abstract class is equivalent to a non-abstract class
- Is often used to create some kind of base class for inheritance that should not be initialised

What is inheritance

Polymorphism

Abstract classes

Interfaces

Interfaces

INTERFACES

```
public interface CanMakeSound {

    public void makeSound();
}
```

```
public class Cow implements CanMakeSound {

    @Override
    public void makeSound() {
        System.out.println("Moooooo");
    }
}
```

- An *interface* is used to define functionality for a class
- If a class *implements* an interface, it must provide implementations for the methods from the interface
- We say that the interface provides a contract for the class

INTERFACES

```
public interface CanMakeSound {

    void makeSound();
}
```

- Methods in an interface are automatically public
- The keyword public may be omitted

INTERFACES

```
public class Cow implements CanMakeSound, CanTravel {

    @Override
    public void makeSound() {
        System.out.println("Moooooo");
    }

    // ...
}
```

- A class may implement multiple interfaces
- Via @Override, it is indicated that a class redefines a method present in an interface or in a super class
  - The class replaces (*overrides*) the method from the interface or super class with new behavior

INTERFACES

```
public static void main(String[] args) {
    CanMakeSound car = new Car();
    CanMakeSound cow = new Cow();

    System.out.println(car.makeSound());
    System.out.println(cow.makeSound());
}
```

- An interface is a valid type for a variable
  - You can cast upwards and downwards
- Working with an interface as an intermediate step allows you to use functionality without worrying about how it is implemented
  - The interface provides an abstraction for the implementation

INTERFACES

```
public static void findShortestDistance(CanTravel traveller) {
    // ...
}
```

- You can use an interface anywhere you can use other types
- So also, for example, as a parameter for a method
  - This is very powerful if you only expect certain behavior in your method but do not care about how that behavior is implemented
  - Eg, anything that can travel can be used to find the shortest distance, the specific class does not matter

EXERCISE @CLASS

- exercise3
- exercise4



EXERCISE @HOME

- exercise5



