2025 — 2026

1TI – 1ACS



UCLL HOGESCHOOL

Back-End Development

Java Classes

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

CONTENTS

- Classes and instances
- Definition of a class
- Working with classes

Classes and instances

Definition of a class

Working with classes

Classes and instances

- What is a class
- What is an instance
- Class vs instance

WHAT IS A CLASS

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

- Why use classes?
  - Break down code into smaller pieces
  - Write clearer programs that feel intuitive

- We have already used externally programmed classes such as
  - The String class (built-in part of Java)
  - The System class to print to console (built-in part of Java)

WHAT IS A CLASS

- Here is an initial example
  - The class "User" has two variables ("name" and "age") that keep track of data
  - The class has methods that can be used to manipulate and query the data

```
public class User {

    private String name;
    private int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void celebrateBirthday() {
        age = age + 1;
    }

    public int getAge() {
        return age;
    }
}
```

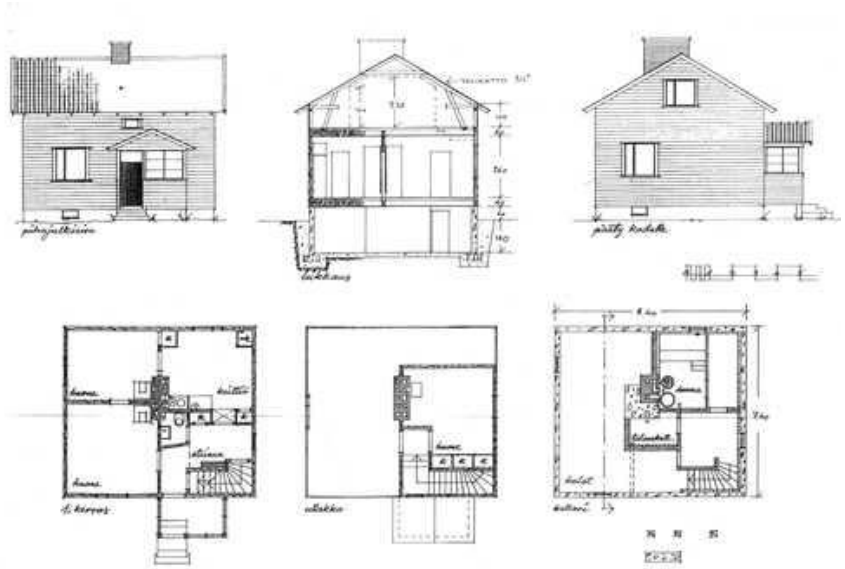
WHAT IS AN INSTANCE

```
public static void main(String[] args) {
    User julie = new User("Julie", 24);

    julie.celebrateBirthday();
    System.out.println(julie.getAge());
}
```

- You can create an *instance* (also called an *object*) of the class using the keyword *new*
- You can store this object in a variable
  - The type of the variable is the class
  - The name of the variable can be chosen freely

CLASS VS INSTANCE



Class — Blueprint



Instance — Specific Realisation

CLASS VS INSTANCE

```java
public class User {

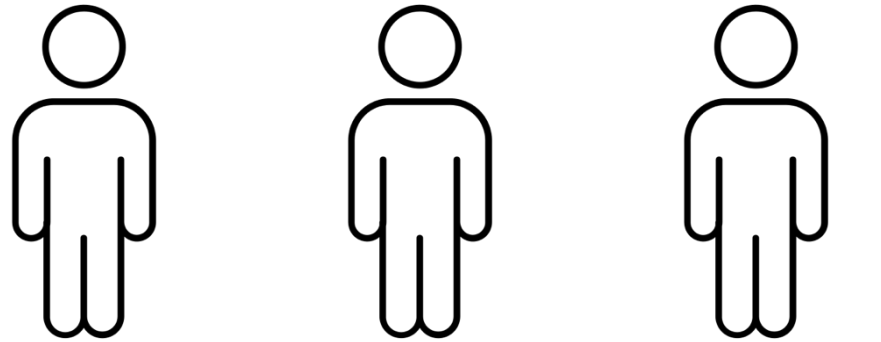
    private String name;
    private int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void celebrateBirthday() {
        age = age + 1;
    }

    public int getAge() {
        return age;
    }
}
```

new



Julie - 24 Emma - 42 Tom - 13

Class — Blueprint

Instance — Specific Realisation

EXERCISE @CLASS

- exercise1



Classes and instances

Definition of a class

Working with classes

Definition of a class

- Fields
- Constructor
- Methods
- Encapsulation

DEFINITION OF A CLASS

```
public class User {

}
```

- The name of a class is written in PascalCase (camelCase but starting with a capital letter)
- A class is defined in a separate file
  - The name of the file is <class-name>.java
- File name and class name must match!

DEFINITION OF A CLASS — FIELDS

```
public class User {

    private String name;
    private int age;

    // ...
}
```

- The variables of a class are defined under the class name
  - In the context of a class, these variables are called fields, attributes or instance variables
- Fields have a certain *visibility* that determines where they can be used
- This visibility is indicated via *access modifiers*

DEFINITION OF A CLASS — FIELDS

```
public class User {

    public String name;
    private int age;

    // ...
}
```

```
public static void main(String[] args) {
    User julie = new User("Julie", 24);

    System.out.println(julie.name);
    System.out.println(julie.age); // Error!!
}
```

- *public* fields are accessible inside the class and outside the class
- *private* fields are only accessible inside the class

DEFINITION OF A CLASS — CONSTRUCTOR

```
public class User {

    // ...

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // ...
}
```

```
public static void main(String[] args) {
    User julie = new User("Julie", 24);
}
```

- An object can have a special method that has the same name as the class
- This method is called the *constructor*
- This is the method that gets called when an instance is made with the keyword *new*

DEFINITION OF A CLASS — CONSTRUCTOR

```
public class User {

    public User(String name) {
        this.name = name;
    }

    public User(String name, int age) {
        this(name);
        this.age = age;
    }
}
```

```
public static void main(String[] args) {
    User julie = new User("Julie", 24);
    User tom = new User("Tom");
}
```

- A class may define multiple constructors if the parameters are different in type or quantity
  - When using *new*, the appropriate constructor will be chosen based on the parameters
  - This is called *overloading* the constructor
- As a first statement, a constructor may call another constructor via *this*

DEFINITION OF A CLASS — CONSTRUCTOR

```java
public class User {

    // Not writing a constructor leads to the implicit
    // presence of a constructor equivalent with:

    // public User() {
    //
    // }
}
```

```java
public static void main(String[] args) {
    User julie = new User();
}
```

- If there is no constructor, Java provides a default constructor without parameters
- This constructor is no longer available as soon as a single constructor is defined

DEFINITION OF A CLASS — METHODS

```
public class User {

    // ...

    public void celebrateBirthday() {
        age = age + 1;
    }

    public int getAge() {
        return age;
    }
}
```

```
public static void main(String[] args) {
    User julie = new User("Julie", 24);

    julie.celebrateBirthday();
    System.out.println(julie.getAge());
}
```

- Methods allow editing or querying an object's field
- Methods, like fields, have a *visibility* with the same meaning
- Methods are invoked on instances of the class

DEFINITION OF A CLASS — METHODS

```
public class User {

    // ...

    public void move(String address) {
        // ...
    }

    public int move(String street, int number, String town) {
        // ...
    }
}
```

- A class may define the same method multiple times if the parameters are different in type or quantity
  - At compile time, the correct method is chosen based on passed parameters
  - This is called *overloading* the method

DEFINITION OF A CLASS — METHODS

```
public class User {

    // ...

    public void celebrateBirthday() {
        age = age + 1;
    }

    public int getAge() {
        return age;
    }
}
```

```
public static void main(String[] args) {
    User julie = new User("Julie", 24);

    julie.celebrateBirthday();
    System.out.println(julie.getAge());
}
```

- Methods may work with fields of the class and manipulate their value
- Methods may call other methods
- Methods may have a return value

DEFINITION OF A CLASS — METHODS

```
public class User {

    private String name;
    private int age;

    // ...

    public boolean isNameOfUser(String name) {
        return this.name.equals(name);
    }
}
```

```
public static void main(String[] args) {
    User julie = new User("Julie", 24);
    System.out.println(julie.isNameOfUser("Tom"));
    System.out.println(julie.isNameOfUser("Julie"));
}
```

- *this* refers to the instance of the class on which the method is called
- Necessary if a method has a parameter with the same name as a field of the class

EXERCISE @CLASS

- exercise2
- exercise3



DEFINITION OF A CLASS — STATIC METHODS

```
public class User {

    public static boolean sameName(
        User user1, User user2) {
        // ...
    }

    private String name;
    private int age;

    // ...
}
```

```
public static void main(String[] args) {
    User julie = new User("Julie", 24);
    User emma = new User("Emma", 8);

    // ...

    User.sameName(julie, emma);
}
```

- A method may include the keyword *static* in its definition
- A static method does not belong to an instance of the class
  - It can be called without instantiating the class
  - The method does not have access to fields and non-static methods
- A static method cannot be overridden by a subclass (see later)

DEFINITION OF A CLASS — STATIC METHODS

```
public class User {

    public static boolean sameName(
        User user1, User user2) {
        // ...
    }

    private String name;
    private int age;

    // ...
}
```

```
public static void main(String[] args) {
    User julie = new User("Julie", 24);
    User emma = new User("Emma", 8);

    // ...

    User.sameName(julie, emma);
}
```

- When should you use a static method?
  - Simple utility methods that do not use instance data
    - Eg, a class Math with different calculation functions that are all static
  - Code that "belongs" in a class but does not use instance data
    - Eg, a class User with a static method to check a password

EXERCISE @CLASS

- exercise4



DEFINITION OF A CLASS — TO STRING METHOD

```
public class User {

    private String name;
    private int age;

    // ...

    public String toString() {
        return name + ", " + age;
    }
}
```

```
public static void main(String[] args) {
    User julie = new User("Julie", 24);
    System.out.println(julie.toString());
    System.out.println(julie);
}
```

- A class may define a toString() method
- If present, this method is called automatically when printing objects of the class

ENCAPSULATION

- Classes are used to structure a program
  - Defining a class "User" with appropriate fields and methods allows other parts of the program to use the class and methods without being concerned with how the method works
  - The internal workings of the class are *encapsulated* towards the rest of the program
- Governing complexity in this way makes it easier to program large programs correctly

Classes and instances

Definition of a class

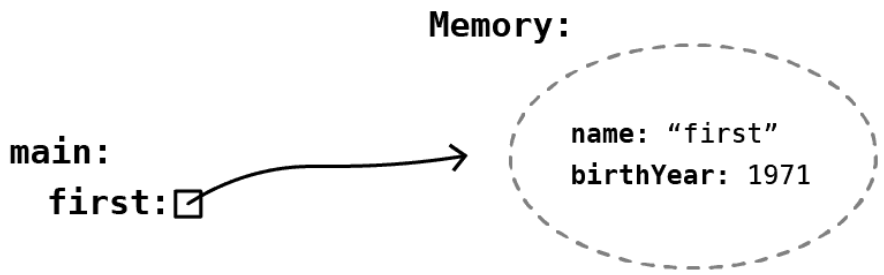
Working with classes

Working with classes

- Classes and variables
- Objects as parameter

CLASSES AND VARIABLES

```
public static void main(String[] args) {
    User julie = new User("Julie", 48);
    User marijke = new User("Marijke", 24);
}
```

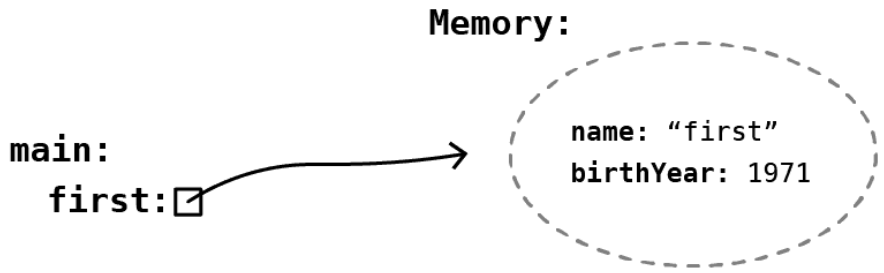


- Variables can have a class as their type
- If a variable has a class as their type, then it is not the data itself that is in the variable, but a *reference* to the data
  - This is different for primitive types, in that case the variable holds the data itself!
  - (The String class is a special case in this respect)

CLASSES AND VARIABLES

```
public static void main(String[] args) {
    User julie = new User("Julie", 48);
    User marijke = new User("Marijke", 24);

    System.out.println(julie == marijke);
    System.out.println(julie.equals(marijke));
}
```



- If you compare variables holding a class using "==" you are comparing references, not contents!
  - Compare primitive types using “==”
  - Compare objects using “.equals()”
    - (For your own classes it is important that an equals method is implemented, see later for details)

CLASSES AND VARIABLES

```
public static void main(String[] args) {
    User julie;
    julie.increaseAge(); // Crash!!

    julie = null;
    julie.increaseAge(); // Crash!!
}
```

- If you define a variable with a class as type but do not initialise it, its value is NULL
  - "NULL" means "no value", this variable has no content
- Calling methods on NULL variables leads to NullPointerExceptions
- You can make a variable NULL via the keyword *null*

OBJECTS AS PARAMETER

- Recall that when a method is called, the value of the variables is copied (pass-by-value)
- Thus, if an object is passed as a parameter, the reference to the data is copied
- This means that in this case the method can adjust the data of the object!

- (The String class is a special case in this respect)

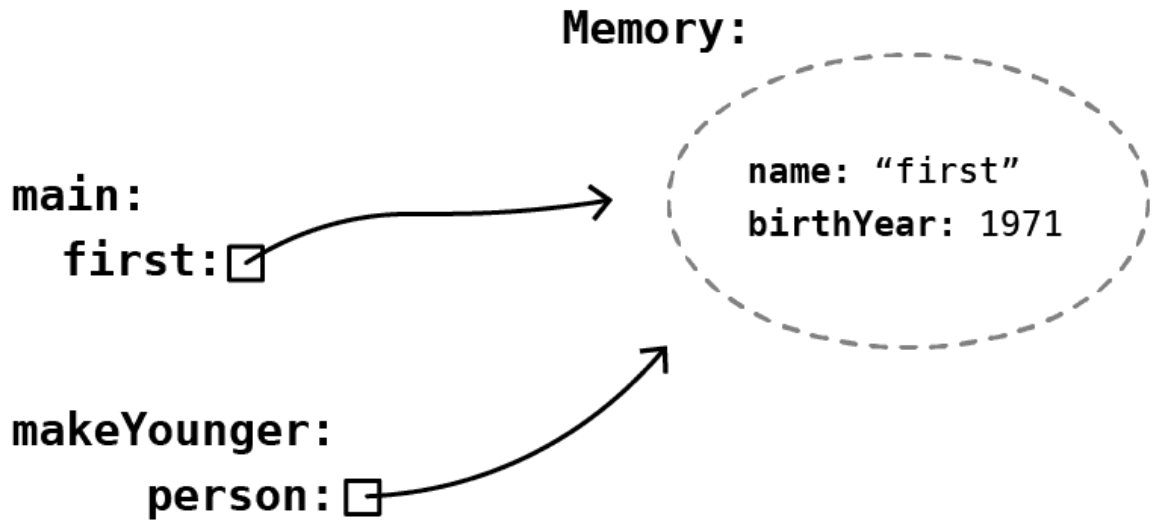
OBJECTS AS PARAMETER

```
public static void main(String[] args) {
    int number = 1;
    increaseInt(number);
    System.out.println(number);

    User julie = new User("Julie", 24);
    increaseAge(julie);
    System.out.println(julie.getAge());
}

public static void increaseInt(int number) {
    number = number + 1;
}

public static void increaseAge(User user) {
    user.increaseAge();
}
```



EXERCISE @CLASS

- exercise5
- exercise6



EXERCISE @HOME

- exercise7
- exercise8

