Unit 3

Definitions of methods and classes

Summary

- Modularization
- Abstraction on operations
- Definition of methods
- Parameter passing
- Execution of a method
- Variables declared inside a method: scope and lifetime
- Abstraction on objects
- Definition of classes
- Access control
- Instance variables
- Methods
- Constructors
- Design of a class

3.1 Modularization

A program can be very long and complex. To manage such a complexity, it is necessary to realize programs in a modular way.

- The program must be structured in autonomous parts, called **modules**.
- The various modules are related to each other through *precise relations*.

A module is characterized by:

- **services exported** to *clients* (these are other modules that make use of such services)
- **an interface**, i.e., the way in which such services are exported
- **imported services**, i.e., services for which the module is a client
- **an internal structure**, i.e., how the module is realized; the internal structure is not of interest for clients.

The use of modularization allows us to realize programs as follows:

1. we define which modules are necessary to solve the requested problem;
2. we define how these modules are related to each other;
3. we develop each module independently from the other modules.

By proceeding in this way we enhance various qualities of a program, which simplify dealing with the program. Specifically:

- readability (or understandability) of the program
- extensibility (i.e., the possibility of extending the functionalities of the program, if necessary)
- reusability of parts of the program for different purposes

3.2 Abstraction

Modularization is tightly coupled with the notion of *abstraction*:

- we construct an abstract model of the problem, based on which we identify the solution
  - we focus on the essential aspects of the problem
  - we ignore those aspects that are not relevant for the objective to achieve
- we select a concrete model that corresponds to the abstract model, and through which we realize the solution to the problem

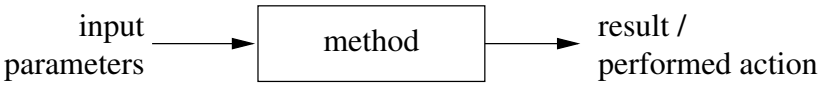
Types of abstraction:

- **Abstraction on operations**: we concentrate on "what" a certain operation has to do, and not on "how" it does what it has to do.
- **Abstraction on objects**:
  - we group similar objects (i.e., objects with the same properties) in classes
  - we establish the relevant properties of the objects, and specifically the operations that the objects support (note that this in turn requires abstraction on operations).

3.3 Abstraction on operations: methods

The abstraction on operations is supported by all current programming languages (Java, C#, C++, C, Pascal, Fortran, Lisp, etc.).

In Java, the abstraction on operations is realized through the notion of **method**. A method can be considered as a black box that takes parameters as input and returns results or performs some actions.



3.4 Methods seen as modules

A method, seen as a module, is characterized by:

- *exported services*: "what" does the method realize
- *interface*: the method header (which specifies the types of the input parameters and of the result, if present)
- *imported services*: other methods or classes used to realize the method
- *internal structure*: Java code describing "how" the method implements the "what" it realizes (note that this is not of interest for the clients of the method, i.e., for whom calls the method)

3.5 Definition of static methods

In Java, the simplest forms of methods are the **static methods**.

Static methods are methods (defined in a class) that do *not* have an invocation object.

Definition of a static method

Syntax:

`header block`

- *`header`* is the header of the static method and has the following form:

  `public static` *`resultType methodName`*`(`*`formalParameters`*`)`

  where
  - `public` indicates that the method being defined is accessible from outside the class (see later)
  - `static` indicates that the method is static (i.e., it has no invocation object)
  - *`resultType`* is the type of the result returned by the method, or `void`, if the method does not return any result
  - *`methodName`* is the name of the method

  - *formalParameters* is a list of parameter declarations (type and name) separated by commas; each parameter is a variable; the list of parameters may also be empty

- *block* is the body of the method containing the statements that will be executed when the method is being called; it has the form:

```
{
  statements
}
```

Semantics:

Defines a static method by specifying its header and body.

- The header indicates:
  - the name of the method,
  - the number and type of its formal parameters,
  - the type of the returned value, if any, and
  - the accessibility of the method from outside the class where it is defined.
- The body of the method specifies the statements that have to be executed when the method is called.
- The formal parameters are used to pass objects, or more generally information, which is used in the body of the method. The formal parameters are used inside the body of the method in the same way as initialized variables (the initialization is done at the moment the method is called, by assigning to each formal parameter the value of the corresponding actual parameter – see later).
- The returned result, if any, is the value of the method-call. If the method does not return any result, then it must not be used to compute a result but to perform side effect on the objects denoted by the formal parameters.

Example:

The method `main()`, used before, is a static method. Such a method has always the form:

```
public static void main (String[] args) {
  ...
}
```

The header of the `main()` method shows that:

- it is a method accessible from outside the class where it is defined (`public`),
- it is a static method (`static`),
- it does not return any result (return type is `void`), and
- it has a parameter of type array of strings (see Unit 7). Up-to now, in our programs, we have never used this parameter.

Note: The definition of a method does not cause its activation.

3.6 Examples of definitions of static methods

Example 1:

```
public static void printGreeting() {
  System.out.println("Good morning!");
}
```

The method `printGreeting()` is a static public method that does not have formal parameters and does not return any result (see header). Its body is constituted by a single statement that prints the string `"Good morning!"`

Example 2:

```
public static void printPersonalGreeting(String firstName, String lastName) {
  System.out.print("Good morning ");
  System.out.print(firstName);
  System.out.print(" ");
  System.out.print(lastName);
  System.out.println("!");
}
```

The method `printPersonalGreeting()` is a static public method that has two formal parameters `firstName` and `lastName` of type `String` and does not return any result (see header). Its body is constituted by a sequence of statements that print respectively the string `"Good morning!"`, the value of the formal parameter `firstName`, a blank, the value of the formal parameter `lastName`, and finally the string `"!"`.

Note that the formal parameters are used inside the body of the method exactly as already initialized local variables.

Example 3: Realize a static method `printInformalGreeting()` that takes as input a string representing a name and prints the string `"Ciao "` followed by the name passed as a parameter and followed by the string `"!"`.

Solution:

```
public static void printInformalGreeting (String name) {
  System.out.println("Ciao " + name + "!");
}
```

3.7 Result of a method: the `return` statement

If a method must return a result, then it must contain a `return` statement.

When the `return` statement is executed inside a method, it causes the termination of the method and it returns the result of the method to the client module (i.e., the part of the program where the method was called).

The syntax of the `return` statement is as follows:

```
return expression;
```

where ***expression*** must be an expression whose value is compatible with the type of the result declared in the method header.

Example:

```
public static String personalGreeting(String firstName, String lastName) {
  return "Good morning " + firstName + " " + lastName + "!";
}
```

If the result type is `void`, the `return` statement can be omitted, or it can simply be used to interrupt the execution of the method. Since in this case it is not necessary to return a result, the syntax in this case is as follows:

```
return;
```

Note: The execution of the `return` statement always terminates the method, even if there are additional statements that follow.

3.8 Example: use of static methods defined in the same class

The following program shows the use of static methods defined in the same class.

```
import javax.swing.JOptionPane;

public class PrintGreetings {

  public static void printGreeting() {
    System.out.println("Good morning!");
  }
```

```
  public static void printPersonalGreeting(String firstName, String lastName) {
    System.out.print("Good morning ");
    System.out.print(firstName);
    System.out.print(" ");
    System.out.print(lastName);
    System.out.println("!");
  }

  public static void printInformalGreeting (String name) {
    System.out.println("Ciao " + name + "!");
  }

  public static String personalGreeting(String firstName, String lastName) {
    return "Good morning " + firstName + " " + lastName + "!";
  }

  public static void main(String[] args) {
    printGreeting();
    String fn = JOptionPane.showInputDialog("First name");
    String ln = JOptionPane.showInputDialog("Last name");
    printPersonalGreeting(fn,ln);
    printInformalGreeting(fn);
    JOptionPane.showMessageDialog(null,personalGreeting(fn,ln));
    System.exit(0);
  }
}
```

Note: The static methods defined in the class `PrintGreetings` are called by the `main()` method of `PrintGreetings` without preceding them by the name of the class. This is possible since the methods belongs to the same class as `main()`.

3.9 Example: use of static methods defined in another class

Let us group now the same methods in a different class.

```
public class Greetings {

  public static void printGreeting() {
    System.out.println("Good morning!");
  }

  public static void printPersonalGreeting(String firstName, String lastName) {
    System.out.print("Good morning ");
    System.out.print(firstName);
    System.out.print(" ");
    System.out.print(lastName);
    System.out.println("!");
  }

  public static void printInformalGreeting (String name) {
    System.out.println("Ciao " + name + "!");
  }

  public static String personalGreeting(String firstName, String lastName) {
    return "Good morning " + firstName + " " + lastName + "!";
  }
}
```

Example of client:

```
import javax.swing.JOptionPane;

public class GreetingsClient {
  public static void main(String[] args) {
    Greetings.printGreeting();
    String fn = JOptionPane.showInputDialog("First name");
    String ln = JOptionPane.showInputDialog("Last name");
    Greetings.printPersonalGreeting(fn,ln);
    Greetings.printInformalGreeting(fn);
    JOptionPane.showMessageDialog(null, Greetings.personalGreeting(fn,ln));
    System.exit(0);
  }
}
```

Note that in the method `main()` of `GreetingsClient` we need to add in front of the calls to the static methods the name of the class where they are defined.

Note: The class `Greetings` can be considered a simple **library** that realizes different greeting functionalities. We will see later the predefined class `Math`, which is a library of the most commonly used mathematical functions on reals, constituted by static methods that realize the functions.

3.10 Parameter passing

As said, the definition of a method contains in the header a list of **formal parameters**. Such parameters are used in the same way as variables inside the body of the method.

The call of a method contains the parameters that have to be used as arguments for the method. Such parameters are called **actual parameters**, to distinguish them from the formal parameters appearing in the header of the method definition.

When a method is called and thus **activated**, the actual parameters have to be *bound* to the formal parameters. In general, there are various ways of establishing such a binding. In Java, there is just one way: the so called *call by value*.

Let `pa` be an actual parameter in a method call, and `pf` the corresponding formal parameter in the header of the method definition: to bind `pa` to `pf` by value means to do the following, when the method is activated:

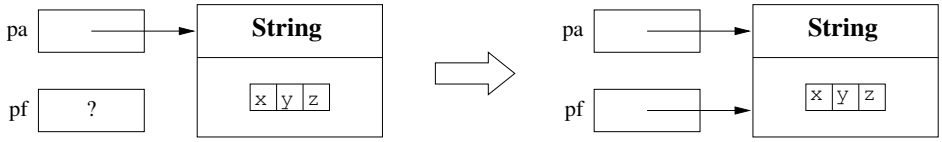
1. The actual parameter `pa` is evaluated (notice that `pa` is in general an expression).
2. A memory location is associated to the formal parameter `pf`.
3. The value of `pf` (i.e., the corresponding memory location) is initialized with the value computed for `pa`.

In other words, the formal parameter `pf` behaves exactly as a local variable created the moment the method is called, and initialized with the value of the corresponding actual parameter `pa`.

At the end of the execution of the body of the method, the memory location reserved for the formal parameter is freed and the value stored in it is lost.

Note: The value of a variable that appears in the expression `pa` is not changed by the execution of the method. Note, however, that if such a value is a reference to an object, then the method can in fact change the object denoted by the reference (see later).

The following figure shows an example of parameter passing for the case where the parameter is a reference to an object. The case of parameters that are of a primitive data type will be described in Unit 4.



3.11 Execution of a method

Consider the following method definition:

```
public static String duplicate(String pf) {
  return pf + ", " + pf;
}
```

Consider then the following `main()` method:

```
public static void main(String[] args) {
  String s;
  s = duplicate("pippo" + "&" + "topolino");
  System.out.println(s);
}
```

Let us analyze in detail what happens when the statement containing the call to the `duplicate()` method is executed:

1. *The actual parameters are evaluated.*

   In our case, the actual parameter is the expression `"pippo" + "&" + "topolino"` whose value is the string `"pippo&topolino"`.
2. *The method to be executed is determined* by considering the name of the method, and the number and the types of the actual parameters. A method with a signature corresponding to the method call has to be found: the name of the method must be the same as in the call, and the formal parameters (i.e., their number and types) must correspond to the actual parameters.

   In our case, the method we are looking for must have the signature `duplicate(String)`.
3. *The execution of the calling program unit is suspended.*

   In our case, it is the method `main()`.
4. *Memory is allocated* for the formal parameters (considered as variables) and for the variables defined in the method (see later).

   In our case, memory is allocated for the formal parameter `pf`.
5. *Each formal parameter is initialized to the value of the corresponding actual parameter.*

   In our case, the formal parameter `pf` is initialized to the reference to the object representing the string `"pippo&topolino"`.
6. *The statements in the body of the called method are executed*, starting from the first one.
7. *The execution of the called method terminates* (either because the `return` statement is executed, or because there are no more statements to execute).

   In our case, the statement `return pf + ", " + pf;` is executed.
8. *The memory for the formal parameters and the local variables is freed*, and all information contained therein is lost.

   In our case, the memory location corresponding to the formal parameter `pf` is freed.
9. *If the method returns a result*, then the result becomes the value of the expression returned by the method invocation in the calling program unit.

   In our case, the result is `"pippo&topolino, pippo&topolino"`.
10. *The execution of the calling unit continues* from the point where it was suspended by the method call.

    In our case, the value `"pippo&topolino, pippo&topolino"` is assigned to the variable `s`.

3.12 Example: modification of an object done by a method

The following program shows what happens when passing parameters that are references to objects.

```
public class Parameters{
  public static void changeValueS (String s) {
    s = s.concat("*");
  }

  public static void changeValueSB (StringBuffer sb) {
    sb.append("*");
  }
```

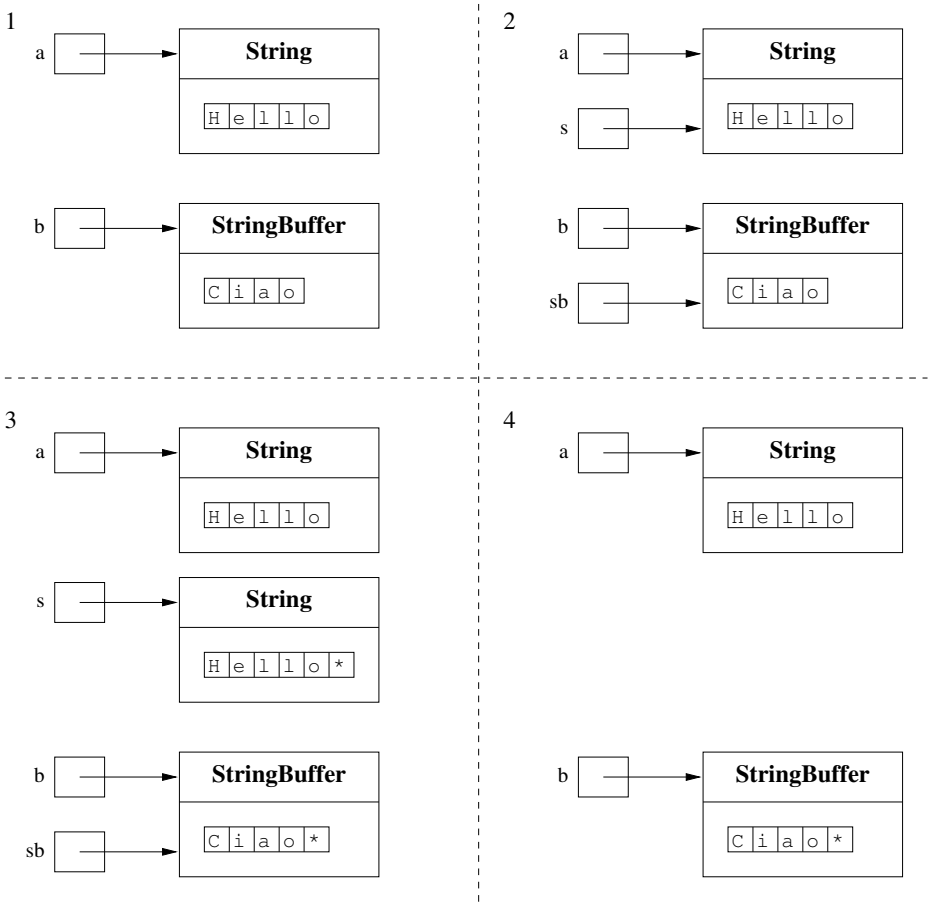
```
    public static void main(String[] args) {
      String a = "Hello";
      StringBuffer b = new StringBuffer("Ciao");

      System.out.println("String a = " + a);
      changeValueS(a);
      System.out.println("String a = " + a);

      System.out.println("StringBuffer b = " + b.toString());
      changeValueSB(b);
      System.out.println("StringBuffer b = " + b.toString());
    }
  }
```

The result of the execution of the program is the following:

```
String a = Hello
String a = Hello
StringBuffer b = Ciao
StringBuffer b = Ciao*
```

The actual parameters `a` and `b` are bound by value to the corresponding formal parameters `s` and `sb`, hence their value (i.e., the reference to the object) is not modified by the execution of the method. However, this does not mean that the state of the object they refer to cannot change (as shown by the example).

The reason why the state of the object referenced by `b` changes, while for `a` this is not the case, is not a direct consequence of parameter passing (note that the parameter is passed in the same way for `a` and `b`). The change depends on the use of the `append()` method, which modifies the state of the object on which it is invoked (i.e., `sb`, which refers to the same object as `b`, whereas the `concat()` method does not modify the state of the object `s`, and hence `a`).



3.13 Local variables

The body of a method can contain variable declarations. Such variables are called *local variables*. In fact, all variables that we have used till now were local variables, since they were declared in the body of a method, namely the `main()` method. Hence, they are used as we have seen in Unit 2.

Here, we concentrate on two fundamental aspects related to variables:

- **scope** (which is a static notion, i.e., it depends on the program text)
- **lifetime** (which is a dynamic notion, i.e., it depends on the execution of the program)

Note: In Java, it is also possible to define *global variables* for a class. A global variable is defined inside the class, but outside any method, and is qualified as `static`. In this course we will not make use of global variables.

3.14 Scope of local variables

The **scope** of a variable is the *region of a program in which the variable is visible*, i.e., in which it is accessible by its name and can be used.

In Java, the scope of a local variable is the body of the method in which it is declared. In other words, the variable is visible in the body of the method where its declaration appears, but it is not visible on the outside the method.

Clearly, as we have already said in Unit 2, a variable cannot be used in the body of a method before it has been declared.

Notes:

- In fact, a more general scope rule holds: the scope of a local variable extends from the point of its declaration to the end of the block that encloses it. A *block* is a statement of the form {...} (see later). Hence, a local variable is visible in the block in which it is declared (including sub-blocks, if present), but is not visible outside that block.
- In Java, the scope of a variable is a completely static notion. Indeed, it can be determined by analyzing the structure of the program, without considering its execution. Most current programming languages support such a notion of *static scope*.
- If follows that the notion of scope is *relevant at compile time*.

3.15 Example: scope of local variables

Consider the following program.

```
public class Visibility {

  public static String duplicate(String s) {
    String t = s + ", " + s;
    return t;
  }

  public static void print1() {
    System.out.println(a);   //ERROR: a is not defined
  }

  public static void print2() {
    System.out.println(t);   //ERROR: t is not defined
  }

  public static void main(String[] args) {
    String a = "Ciao";
    a = duplicate(a);
    print1();
    print2();
    System.out.println(a);
```

```
  }
}
```

During program compilation the compiler will signal two errors:

1. In the `print1()` method, the variable `a` is not visible (since it is defined in the `main()` method).
2. In the `print2()` method, the variable `t` is not visible (since it is defined in the `duplicate()` method).

3.16 Lifetime of local variables

The **lifetime** of a variable is the *time during which the variable stays in memory and is therefore accessible during program execution.*

The variables that are local to a method are created the moment the method is activated (exactly as formal parameters) and are destroyed when the activation of the method terminates.

More precisely, when the method is activated, a block of memory cells, called *activation record*, is allocated, which contains all local variables and formal parameters of the current method call. The activation record is used during the method execution, and is then removed at the end of the execution. When the activation record is removed, the memory locations for the local variables and for the formal parameters are destroyed and the values they contain are lost.

When the method is activated again, a new activation record is allocated, with new memory locations that have nothing to do with the ones of the previous activations. Hence, at each method activation the memory locations for local variables and formal parameters are created anew, and these memory locations are in general different from the ones of previous activations. It follows, that the values of local variables and of formal parameters are **not kept** from one method call to the next.

Note: The notion of lifetime of a variable is *relevant at execution time.*

3.17 Overloading of methods

As said, Java distinguishes methods based on their whole signature, and not only on their name. Hence, we can define in the same class more than one method with the same name, as long as these methods differ in the number or type of their formal parameters (note: the name of the formal parameters is not relevant for the distinction). This feature is called *overloading* of methods.

Example:

```
public class Greetings2 {

  public static void printGreeting() {
    System.out.println("Hello!");
  }

  public static void printGreeting(String name) {
    System.out.println("Hello " + name + "!");
  }
}
```

- The call of the method `printGreeting()` without actual parameters will activate the first `printGreeting()` method (the one without formal parameters), which will print the string `"Hello!"`.
- The call of the method `printGreeting()` with one actual parameter of type `String` will activate the second `printGreeting()` method, which will print the string `"Hello "`, followed by the string passed as a parameter, followed by `"!"`.

3.18 Abstraction on objects

Abstraction on objects is realized as follows:

- We group similar objects (i.e., objects with the same properties) into classes.
- We establish the properties that are relevant for the objects, and in particular, the operations that they support (note that this requires to perform an abstraction on operations).

The ability to support in a very advanced way the abstraction on objects is the fundamental feature of all object-oriented programming languages (such as Java, C++, C#, etc.). In these languages, such a form of abstraction is supported by the ability to define *classes* directly at the programming language level.

In Java, the **definition of a class** is characterized by:

- the *name* of the class, which identifies the class itself, and hence identifies the type of its instances;
- the *instance variables* (also called *data fields*), which allow us to store data inside the objects;
- the *(instance) methods* (also called *operation fields*), which can be invoked on the objects of the class to perform operations on them.

Note: The instance variables and the methods represent the properties of the objects of a Java class.

Moreover, through suitable **access modifiers**, we can specify:

- which fields should be visible to the outside the class, i.e., to the clients of the class – such fields are called *public*;
- which fields should be hidden to the clients since they are not relevant for them – such fields are called *private*.

3.19 Classes seen as modules

A class, seen as a module, is characterized by the following features (we assume that no instance variable is public):

- *exported services*: the public methods, i.e., the methods visible outside the class;
- *interface*: the headers of the public methods;
- *imported services*: other methods or classes used to realize the representation of the objects and of the methods of the class;
- *internal structure*: representation of the objects and realization of the methods of the class.

Note: The abstraction on objects makes use of the abstraction on operations.

3.20 Definition of a class

Definition of a class

Syntax:

```
public class Name {
  field-1
  ...
  field-n
}
```

- `Name` is the name of the class
- `field-1` ... `field-n` are the *fields* of the class, which represent its properties. Each `field-i` can be either a *data field* or an *operation field*:
  - a *data field* (or *instance variable*) is a variable declaration (see later)
  - an *operation field* (or *method*) is a method definition (see later)

Each field is qualified with an *access modifier*, which determines its visibility outside the class (see later).

Semantics:

Defines a class.

- The data fields (or instance variables) are used to represent the internal structure of the objects of the class.
- The operation fields (or methods) are used to realize the functionalities of the class.

The details will be shown later.

3.21 Example of a class definition

We want to realize a Java class to represent persons. The properties of interest for a person object are the *name*, defined once and for all, and the *residence*, which may change.

Let us define a Java class `Person` to represent persons.

```
public class Person {
  //instance variables (data fields)
  private String name;
  private String residence;

  //methods (operation fields)
  public String getName() {
    return name;
  }
  public String getResidence() {
    return residence;
  }
  public void setResidence(String newResidence) {
    residence = newResidence;
  }
}
```

The definition of the class `Person` consists of the following elements:

- the name of the class, i.e., `Person`;
- two `private` data fields (or instance variables) of type `String`, namely `name` and `residence`;
- three `public` fields, each of which is a method definition, namely `getName()`, `getResidence()` and `setResidence()`.

The keywords `public` and `private` specify which fields are public and which are private (see later).

- Name and residence of persons are *represented* each by an instance variable of type `String` in the class `Person`. Since these variables are private, they cannot be accessed from outside the class.
- Two of the methods defined in the class, `getName()` and `getResidence()`, return respectively the name and the residence of a person.
- The method `setResidence()`, instead, allows us to change the residence of the person represented by the invocation object for the method.

When a method, such as `setResidence()`, modifies the object on which it is called we say that it has a **side-effect** (see, e.g., the methods of the class `StringBuffer`). In general, the decision whether a method of a class should have a side-effect or not is a design choice that has important consequences on the way in which the class must be used by its clients.

Note: The definition of a class has to be saved in a file with the same name as the class and extension `.java`. For example, the definition of the class `Person` must be saved in a file called `Person.java`.

Note: In the definition of a class, the order of the fields (instance variables and methods) is irrelevant.

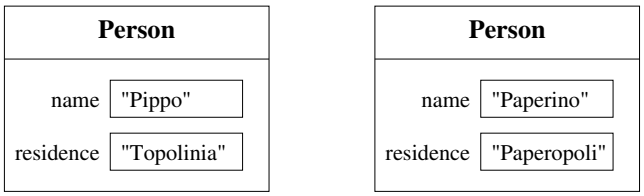
3.22 Instance variables

Instance variables are variables defined in a class, but outside the body of methods. The declaration of an instance variable is similar to the declaration of a local variable of a method, but:

1. the variable is defined inside the class, but outside all methods;
2. the variable is preceded by an access modifier (usually `private`);
3. the variable is *always initialized* when the object is created, either implicitly (to a default value), or explicitly by the constructor (see later).

   Note: This is different from local variables, which are not necessarily initialized when the associated memory location is created.

The instance variables are associated to the single objects and not to the entire class. This means that each object has its own instance variables. Two different objects have different instance variables.



3.23 Use of a defined class

A class defined by the programmer is used exactly in the same way as a predefined class (e.g., `String`).

```
public class ClientClassPerson {
  public static void main(String[] args) {
    Person p1;
    p1 = new Person();
    p1.setResidence("Roma");
    System.out.println(p1.getResidence());
  }
}
```

The class `ClientClassPerson` is a client of the class `Person`, since it makes use of such a class. The client defines the method `main()` (the method of the program that is called first) in which:

1. we define a variable, local to `main()`, of type `Person` (or, more precisely, of type reference to an object that is an instance of `Person`);
2. we create a new object of the class `Person`, and we assign to `p1` a reference to it;
3. we call the method `setResidence()` of the class `Person` on the object denoted by `p1`, and pass to the method the actual parameter `"Roma"`; note the use of the **selection operator** "." to select the (public) fields of the class (in this case, the method `setResidence()`);
4. finally, we call the method `getResidence()` on `p1` to print the residence of the object denoted by `p1`.

Note: The class `ClientClassPerson` must be saved in a file called `ClientClassPerson.java`. The file must be placed in the same directory as the file containing the class `Person`, in order to avoid problems during the compilation or the execution of the class `ClientClassPerson`. We could overcome this limitation by making use of so called *packages*, but we will not deal with packages in this course.

3.24 Controlling the access to the fields of a class

The **access modifiers** `public` and `private` have the following meaning:

- `public` indicates that the method / instance variable *is visible* outside the class, i.e., by the clients of the class.
- `private` indicates that the method / instance variable *is not visible* outside the class, and hence can only be used inside the class itself and not by its clients.

Example:

```
public class ClientClassPerson2 {
  public static void main(String[] args) {
    Person p1;
    p1 = new Person();
    p1.setResidence("Roma");
                //OK! the field setResidence is public
    System.out.println(p1.getResidence());
                //OK! the field getResidence is public
    System.out.println(p1.residence);
                //ERROR! the field residence is private
  }
```

```
}
```

This example shows a client that tries to access the public and private fields of the class `Person`. While the access to the public fields, `setResidence()` and `getResidence()`, is allowed, the access to the private field, `residence`, causes an error at compile time.

3.25 Scope of instance variables

Instance variables are always visible in all methods of the class. They always refer to the invocation object.

Example: In the statement `return name;` the instance variable `name` is an instance variable of the invocation object for the method.

The public instance variables are visible outside the class, and they can be accessed through a reference to the object to which they belong, by means of the field selection operator ".".

Example: If we had defined the class to represent persons as follows:

```
public class Person2 {

  //instance variables (data fields)
  private String name;
  public String residence;        //residence is declared public

  //methods (operation fields)
  public String getName() {
    return name;
  }
  public String getResidence() {
    return residence;
  }
  public void setResidence(String newResidence) {
    residence = newResidence;
  }
}
```

then we could directly access the instance variable `residence`, as shown by the following client:

```
public class ClientClassPerson2 {
  public static void main(String[] args) {
    Person2 p1;
    p1 = new Person2();
    p1.setResidence("Roma");
            //OK! the field setResidence is public
    System.out.println(p1.getResidence());
            //OK! the field getResidence is public
    System.out.println(p1.residence);
            //OK! the field residence is public
  }
}
```

Note: Typically, the instance variables must be declared `private`, to hide from the clients the representation of the objects of the class. This leaves the freedom to change such a representation without the need to modify the clients.

3.26 Lifetime of instance variables

The lifetime of an instance variable coincides precisely with the lifetime of the object of which it is part. The instance variables are created the moment the object containing them is created, and they can be used as long as the object can be accessed. The creation of the object denoted by a variable is done by the run-time support (the Java Virtual Machine), by allocating, together with the object, the memory that is necessary to store the values of its instance variables.

Through the mechanism of *garbage collection*, the run-time support automatically destroys an object (and frees the memory it takes up) when there are no more references to the object, and hence the object cannot be accessed any more.

3.27 Rules for accessing the fields of a class

Typically, the access to the fields of a class is done as follows:

- The methods that correspond to functionalities of the class that are of interest for the clients are declared as `public`.
- The instance variables and the auxiliary methods, i.e., those methods that help to realize the methods representing the functionalities of interest, are declared as `private`.

In this way, the methods that correspond to functionalities of interest for the clients are visible outside of the class, while the instance variables and the auxiliary methods that are needed to support those functionalities, but that are not of interest for the clients, are visible only inside the class.

The set of `public` fields of a class is called the **public interface** of the class.

Note: In Java, there are two additional access modalities, namely `protected` and "visible in the package" (the latter is obtained by omitting the access modifier). These will not be discussed in this course, but will be dealt with in subsequent programming courses.

3.28 Definition of methods

The **definition of a method** is similar to the definition of a static method that we have already seen, but in the header of the method the `static` keyword does not appear. This indicates that the method requires an invocation object.

Hence, the definition of a method is constituted by:

header block

- *header* is the header of the method and has the following form:

  `public` *resultType methodName*`(`*formalParameters*`)`

  where (as for the static methods)
  - `public` indicates that the method being defined is accessible from outside the class
  - *resultType* is the type of the result returned by the method, or `void`, if the method does not return any result
  - *methodName* is the name of the method
  - *formalParameters* is a list of parameter declarations (type and name) separated by commas; each parameter is a variable; the list of parameters may also be empty
- *block* is the body of the method containing the statements that will be executed when the method is being called; as for static methods, it has the form:

  ```
  {
    statements
  }
  ```

3.29 The implicit formal parameter `this`

All the instance (i.e., non-static) methods have an implicit formal parameter denoted by `this`. Such a parameter denotes the invocation object. In other words, when the method is called, `this` is bound to (the reference to) the invocation object, which thus acts as an actual parameter.

The parameter `this` is used to access the instance variables and the methods of the invocation object. Note that, in general, we can omit `this`, as we have done till now. Indeed, Java inserts it automatically whenever we use an instance variable or an instance method of the class.

Example: The definition of the class `Person` given below has exactly the same meaning as the class `Person` that we have already seen.

```
public class Person {
  //instance variables (data fields)
  private String name;
  private String residence;

  //methods (operation fields)
  public String getName() {
    return this.name;
  }
  public String getResidence() {
    return this.residence;
  }
  public void setResidence(String newResidence) {
    this.residence = newResidence;
  }
}
```

Note: We cannot assign a value to the formal parameter `this`. Note that, if we could, it would mean that we would actually change the invocation object of the method.

3.30 Use of `this`

Typically, `this` is used when there is a local variable (or a formal parameter) declared inside the method with the same name as an instance variable, and we want to distinguish between the instance variable and the local variable. Indeed, if we declare a local variable using the same identifier as for an instance variable, then the name of the local variable hides the name of the instance variable, and we need to explicitly use `this` to denote the instance variable (the implicit use of `this` is blocked).

Example:

```
public class Person {
  private String name;
  private String residence;

  public String getName() {
    return name;
  }
  public String getResidence() {
    String residence;
       // the local variable masks the instance variable with the same name
    residence = this.residence;
       // this is used to distinguish the instance var from the local var
    return residence;
       // here we are referring to the local variable
  }
  public void setResidence(String residence) {
    this.residence = residence;
       // this is again used to distinguish the instance var from the local var
  }
}
```

3.31 Constructors

With what we have seen till now we have no means to initialize with a suitable value the field `name` of a `Person` object. We do not know how to let an object correspond, e.g., to John Smith. The instance variable name should have the value `"John Smith"`, but this variables is `private`, and hence the following statement would be wrong:

```
Person p = new Person();
p.name = "John Smith";   // ERROR! name is declared private
```

To make it possible to explicitly initialize `private` instance variables of objects, we have to use constructors. A **constructor** is simply a (non `static`) method of a class that has the same name as the class and does **not** have an explicit return value (not even `void`).

For example, let us realize a constructor for the class `Person` that takes as arguments the name and the residence of the person to create.

```
public class Person {
  ...
  // constructor name-residence
  public Person(String n, String r) {
    name = n;
    residence = r;
  }
  ...
}
```

3.32 Invocation of a constructor

A constructor is automatically called by the run-time support (the Java Virtual Machine) when an object is created using the `new` operator. For example, with the following code fragment

```
Person p = new Person("John Smith", "London");
                // constructor name-residence is called
System.out.println(p.getResidence());
                // prints "London"
```

the run-time support calls the constructor `Person(String,String)`, which creates (i.e., allocates the memory for) an object of the class `Person` and initializes explicitly the fields `name` and `residence` to the values passed as parameters. The reference to the newly created object is then assigned to the variable `p`.

Consider the following code fragment:

```
Person p;                                  // (1)
p = new Person("John Smith", "London");  // (2)
```

In (1), we define a variable `p` of type reference to an object of type `Person`, while in (2) we create a new object `Person`, and we assign the reference to it to the variable `p`.

Note: The `new` operator uses a constructor to create an object, and returns a reference to it. Such a reference can:

- be passed as actual parameter to a method that has a formal parameter of type reference to `Person`.
- be returned as result of a method whose return value is of type reference to `Person`.

Note: It is important that all constructors are declared as public fields of the class. If they were declared private, then any attempt to create an object of the class would generate an error.

3.33 Overloading of constructors

Since Java admits overloading of methods, and a constructor is a special case of a method, it is possible to define several constructors for a class.

For example, we can define a constructor that sets to `null` the residence of persons that are being created.

```
// constructor name
public Person(String n) {
  name = n;
  residence = null;
}
```

We show some examples of how to use the constructors:

```
Person p1 = new Person("John Smith");
                // calling constructor name
```

```
Person p2 = new Person("Tom Jones", "London");
                  // calling constructor name-residence
System.out.println(p1.getName());
                  // prints "John Smith"
System.out.println(p2.getName());
                  // prints "Tom Jones"
```

When we create an object by means of a `new` operation, the compiler determines which constructor to use based on the number and the types of parameters specified in the `new` operation. The run-time support can then call the chosen constructor to create the object.

3.34 Standard constructor

When we create an object of a class that does not contain any constructor definition (such as the first version of the `Person` class), the so-called **standard constructor** is called.

- The standard constructor is a *constructor without arguments* generated automatically by the compiler for all those classes that do not contain any constructor definition.
- It leaves the instance variables initialized to their *default value*, which is the value assigned automatically by the run-time support when the memory location associated to the variable is being reserved.
- The standard constructor is automatically inhibited by the compiler when the definition of any constructor (with or without arguments) is explicitly present in the class. In particular, the programmer can also explicitly define a constructor without arguments that replaces the standard constructor.

For example, for the class `Person`, we could define the following constructor without arguments:

```
public Person() { // constructor without arguments
  name = "John Smith";
  residence = null;
}
```

Note: It does not always make sense to define for a class a constructor without arguments. For example, the definition of a constructor without arguments for the class `Person` can certainly be questioned.

3.35 Design methodology for a class: realization of a class

We present a methodology for designing a class in various steps, which will allow us to realize Java classes in a structured way, by dividing the problem in various sub-problems, and addressing each of them separately. In this way we will be able to deal with the complexity of realizing a class in a simple and effective way.

1. Starting from the specification of a class, *we identify the properties and the services of the class to realize.*
2. We choose a *representation for the objects of the class*, by identifying the instance variables that are necessary.
3. We choose the *headers of the public methods of the class* (the interface of the class). Note that in this step we are deciding in which way the clients of the class have to use the objects of the class we are realizing.
4. We *realize the body of the public methods*, possibly by introducing auxiliary methods in order to simplify and structure the code.

3.36 Design methodology for a class: clients of the class

Once a class is realized, we should also realize an example client of the class to verify in practice how the class has to be used. Note that, in order to do so, we do not have to know the bodies of the public methods of the class. Indeed, from the point of view of the clients, what matters is **what** the public methods of the class do, **not how** they do it.

This means that we could also anticipate the realization of the clients of the class before we have realized the bodies of the public methods (and hence also before we introduce auxiliary methods).

In practice, after step 3 we can realize the so-called **skeleton of the class**, i.e., the class itself, in which we have just the headers of the public methods, instead of their definitions, and without any private method.

The skeleton of the class is sufficient to realize the clients of the class.

3.37 Example for the design of a class

Specification: Realize a Java class for representing cars. The properties of interest for a car are the plate number, the model, the color, and the person who owns the car. The first two properties cannot be modified, while the third and fourth can. A car has initially no owner. The owner is assigned to a car only later (e.g., when the car is sold).

By analyzing the above specification, we see that we have to realize a class `Car`, whose functionalities are:

- creating an object of the class with the properties plate, model, and color initialized to suitable values, and no owner;
- returning the value of each of the properties plate, model, color, and owner;
- changing the color or the owner.

At this point we are ready to write:

```
public class Car {
  // private representation of the objects: instance variables
  // public methods realizing the requested functionalities
}
```

3.38 Example for the design of a class: representation for the objects

We have to decide how to represent the properties of cars. In this case, it is immediate to choose the representation. We represent the objects of the class `Car` by means of the following instance variables:

- the plate, by means of an instance variable `plate` of type `String`
- the model, by means of an instance variable `model` of type `String`
- the color, by means of an instance variable `color` of type `String`
- the owner, by means of an instance variable `owner` of type `Person`

Note: In our initial examples, the choice of the representation will always be immediate. However, as we go on with the course, we will see that this step can become significantly more complex.

At this point we are ready to write:

```
public class Car {
  // representation of the objects
  private String plate;
  private String model;
  private String color;
  private Person owner;

  // public methods realizing the requested functionalities
}
```

3.39 Example for the design of a class: public interface

We are now ready to choose the interface of the class `Car`, through which the clients can use its objects. In particular, for each functionality we have to define a public method realizing it and determine its header.

The requested functionalities are:

Creation of an object of the class with the properties plate, model, and color, suitably initialized, and no owner.

We know that, to construct an object of a class, we have to use a constructor. Hence, this functionality requires the definition of a constructor; specifically, this constructor must initialize the instance variables that represent plate, model, and color using suitable parameters (note that the first two properties cannot change value anymore). The instance variable `owner`, instead, must be initialized to the non-significant value `null`. The header of the constructor is:

```
public Car(String p, String m, String c)
```

Return of the value of each of the properties plate, model, color, and owner.

For each of the four properties, we define a public method that return the value (to be precise, the reference to the object that represents the value). The headers of these methods are:

```
public String getPlate()
public String getModel()
public String getColor()
public Person getOwner()
```

Modification of the value of the properties color and owner.

To modify the color and the owner we introduce two methods whose header is:

```
public void setColor(String newColor)
public void setOwner(Person newOwner)
```

At this point we can write the skeleton of the class `Car`:

```
public class Car {
  // representation of the objects
  private String plate;
  private String model;
  private String color;
  private Person owner;

  // constructor
  public Car(String p, String m, String c) {
    ...
  }
  // other pubblic methods
  public String getPlate() {
    ...
  }
  public String getModel() {
    ...
  }
  public String getColor() {
    ...
  }
  public Person getOwner() {
    ...
  }
  public void setColor(String newColor) {
    ...
  }
  public void setOwner(Person newOwner) {
    ...
  }
}
```

Note: Since we have introduced a constructor, we cannot use anymore the standard constructor. On the other hand, we are not interested in defining a constructor without arguments, since we need to fix the plate and the model of an object `Car` once and for all the moment it is created.

3.40 Example for the design of a class: realization of the methods

We concentrate now on the various methods and realize their bodies.

We start from the constructor:

```
public Car(String p, String m, String c) {
  plate = p;
  model = m;
```

```
    color = c;
    owner = null;
  }
```

Note: If we omit the statement `owner = null;` the `owner` will anyway automatically be initialized to `null`, which is the default value for references to objects. It is anyway a good programming practice to *explicitly initialize all instance variables*, thus avoiding to make use of the automatic initializations.

We realize the other methods in a similar way.

```
public class Car {
  // representation of the objects
  private String plate;
  private String model;
  private String color;
  private Person owner;

  // constructor
  public Car(String p, String m, String c) {
    plate = p;
    model = m;
    color = c;
    owner = null;
  }
  // other public methods
  public String getPlate() {
    return plate;
  }
  public String getModel() {
    return model;
  }
  public String getColor() {
    return color;
  }
  public Person getOwner() {
    return owner;
  }
  public void setColor(String newColor) {
    color = newColor;
  }
  public void setOwner(Person newOwner) {
    owner = newOwner;
  }
}
```

3.41 Example for the design of a class: a client

Let us realize a client, `CarServices`, of the class `Car`. The class `CarServices` contains two static methods:

- `spray()`, which takes as parameters (a reference to) an object `Car` and (a reference to) an object `String`, representing the (new) color of the car, and modifies the object `Car` by changing its color;
- `registerAlfa147()`, which takes as parameters two objects `String` that represent respectively the plate and the color and returns (a reference to) a new object `Car` whose model is `"Alfa147"` and whose plate and color are specified by the parameters.

We write the class `CarServices`, in a file called `CarServices.java`, as follows:

```
public class CarServices {
  public static void spray(Car car, String color) {
    car.setColor(color);
```

```
  }
  public static Car registerAlfa147(String pla, String col) {
    return new Car(pla, "Alfa147", col);
  }
}
```

Finally, we realize a class `Main`, containing a method `main()`, that uses the class `Car` and the class `CarServices`. This class will be written in a separate file `Main.java`:

```
public class Main {

  // auxiliary method
  private static void printCarData(Car a) {
    System.out.println("Car: " + a.getPlate() + ", "
                               + a.getModel()+ ", "
                               + a.getColor());
  }

  // auxiliary method
  private static void printOwnerData(Car a) {
    System.out.println("Owner: " + a.getOwner().getName() + ", "
                                 + a.getOwner().getResidence());
  }

  public static void main(String[] args) {
    Car a = new Car("313","Fiat 500","Red and Blu");
    printCarData(a);
    Person p = new Person("Paperino", "Paperopoli");
    a.setOwner(p);
    printOwnerData(a);
    CarServices.spray(a, "Maranello Red");
    printCarData(a);
    Car b = CarServices.registerAlfa147("131", "Alfa Red");
    printCarData(b);
    Person c = new Person("Clarabella", "Topolinia");
    b.setOwner(c);
    printOwnerData(b);
  }
}
```

3.42 The `toString()` method

The `toString()` method is a method defined for all objects, independently of the class to which they belong, and even if we do not define it explicitly. The reason for this is that the `toString()` method is defined for the class `Object`, and hence is *inherited* by all Java classes (we will discuss inheritance in more detail in Unit 8).

The `toString()` method has the following header:

```
public String toString()
```

This method is used to transform an object into a `String`. Typically, it is used to construct a string containing the information on an object that can be printed, and we can redefine it for a certain class. If we do not redefine it, the `toString()` method of the class `Object` is used (which prints a system code for the object).

Example:

```
public class TestToString {
  public static void main(String[] args) {
    Person p = new Person("Pippo", "Topolinia");
    System.out.println(p.toString());
  }
```

```
}
```

This program is executed without errors and prints a string on the screen, for example `"Person@601bb1"`, which corresponds to a code defined by the method `toString()` of `Object`.

We can redefine the method `toString()` in the class `Person` in such a way that it returns the name of the person.

```
public class Person {
  ...
  public String toString() {
    return name;
  }
  ...
}
```

Now, the same program `TestToString` prints `"Pippo"`.

3.43 Use of `toString()` in `print()` and `println()`

The predefined class `PrintStream` contains variants of the methods `print()` and `println()`, which we have used till now, that have a formal parameter of type reference to `Object` instead of `String`. These two methods invoke on the parameter of type `Object` the method `toString()`, and then print the resulting string using the printing method that we have already seen for `String`. In practice, this allows us to avoid the explicit use of `toString()` in the argument of `print()` and `println()`.

Example:

```
public class TestToString2 {
  public static void main(String[] args) {
    Person p = new Person("Pippo", "Topolinia");
    System.out.println(p);
    // this is equivalent to System.out.println(p.toString());
  }
}
```

Exercises

Exercise 3.1. Define a class `Book` to handle the information associated to books. The information of interest for a book are: the title, the authors, and the price. The methods of interest are:

- a constructor to create a book object, with title and authors as parameters;
- `printBook()`, which prints the title and the authors of a book on two lines;
- `printBookPrice()`, which prints the title, the authors, and the price of the book;
- `windowBook()`, which shows the title and the authors of a book on two lines in an output window;
- `getTitle()`, which returns the title of a book;
- `getAuthors()`, which returns the authors of a book;
- `getPrice()`, which returns the price of a book;
- `setPrice()`, which sets the price of a book to the integer value passed as parameter (note: use the primitive type `int` for integers – see Unit 4).

Exercise 3.2. Write a Java class implementing an example client for the class `Book` of Exercise 3.1. The example class should perform the following operations:

1. read from the keyboard the title and authors of a first book, and create a corresponding object;
2. read from the keyboard the title and authors of a second book, and create a corresponding object;
3. show the information about the first book;
4. show the information about the second book;
5. read the price of the first book and update the object accordingly;
6. show the information about the first book, including the price.

Exercise 3.3. We want to realize a system for composing messages to send via a cell phone. Each message corresponds to a code. For example, to the code `"ily"` corresponds the complete message `"I love you, darling."`. Define the class `MessageText` to handle messages. The class must have:

- a method to create an object `MessageText`, given the code and the complete message;
- a method to return the code of a message;
- a method to return the text of the message.

Exercise 3.4. A message must contain the number to call, the number of the sender, and the message text. Define a class `Message`, with:

- a method that creates an instance starting from an object of the class `MessageText` and two strings that represent the number to call and the number of the sender;
- a method that creates an instance starting from an object of the class `MessageText` and a string that represents the number of the sender;
- a method that creates an instance starting from an object of the class `MessageText`;
- a method that takes as parameter a phone number, and uses it to update the number to call of the message;
- a method that prints the message.

Exercise 3.5. Write an example program that uses the two classes `MessageText` and `Message`. The program should:

1. inizialize a variable of type `String` that corresponds to the phone number of the sender;
2. read the text and the code of a message, and store them in an object of type `MessageText`;
3. read the phone number of the receiver;
4. create an object of type `Message`;
5. show the information about the object of type `Message`.

Define (and use) an auxiliary static method for reading a phone number from the keyboard. Solve the exercise so as to use all methods that have been defined in the classes `MessageText` and `Message` (possibly by creating and printing various messages).

Exercise 3.6. Modify the classes `Book` (Exercise 3.1) and `MessageText` (Exercise 3.3) by defining static methods to read the input from the keyboard. Test these methods by suitably modifying the example programs.