Introduction to Object-Oriented Programming

Session 1

Alessandro Sgarabottolo

LMU, Master in Financial and Insurance Mathematics

07.04.2025

Contents

Machine model vs Model of the problem

Suppose you want to **solve a problem by using your computer**.

- The first step is to think how to *translate* your problem in terms of the machine, that is, in the end, in terms of the programming language you like to use.
- Of course you want this translation to be as close as possible to your original problem: probably, you don't want to represent your problem as a long sequence of 0s and 1s.
- In other words: you want to represent your problem as closely as possible to the way you think about it.

When using a language close to the machine language, indeed, the programmer must establish an association between the machine model and the model of the problem that is actually being solved.

- This produces programs that are difficult to write and expensive to maintain.
- The process of providing tools for the programmer to represent a problem as close as possible to the way a human thinks and as far as possible to machine language is known as progress of abstraction.

The progress of abstraction: from machine language to imperative languages

- Machine language (sequences of 0s and 1s) is the lowest possible level of abstraction: the closest to the machine. It represents the most fundamental level of information stored in a computer system.
- Assembly language is a small abstraction of the underlying machine: there is still a very strong correspondence between the instructions in the language and the architecture's machine code instructions.
- *Imperative* languages (for example Fortran, BASIC, and C) are abstractions of assembly language. However, a problem is solved by a sequence of instructions. The programmer is still required to think in terms of the structure of the computer rather than in terms of the structure of the problem.

Example

Assume you want to store the number 42 in the register `A`, 14 in the register `B` and then sum the number in the register `A` and the one in the register `B`.

- In machine language this would read something like 1452, 42, 1737, 14, 4541 (translated then into binary numbers!):
  - The number 1452 corresponds to `load the following number into the register A`;
  - The number 42 is the value we want to store in `A`;
  - The number 1737 corresponds to `load the following number into the register B`;
  - The number 14 is the value we want to store in `B`;
  - The number 4541 corresponds to `sum the value in A with the value in B`;
- In assembly language it would read like:

```
LDA 42
LDB 14
ADD.
```

- In an imperative language it would be

```
A = 42
B = 14
add(A, B).
```

Object oriented programming: a step further towards abstraction

- The object-oriented approach goes a step further towards abstraction: it provides tools for the programmer to divide the problem into sub-elements and represent these elements in the machine. In other words, object-oriented programming allows users to program in a similar way as they work with real-life entities.
- The sub-elements are represented by objects.
- The idea is that the program can be adapted to the space of the problem by adding new types of objects, i.e., classes. In this sense, when you read the code describing the solution, you are reading words that also express the problem.
- One of the challenges of object-oriented programming is to create a one-to-one mapping between elements in the problem space and types of objects in the solution space.
- The basic concept of OOP is to create types of objects, re-use them throughout the program creating their instances, and manipulate these objects to get results.

Classes and objects

- A class is a kind of fancy type of variable: it describes how a thing is build, what are its characteristics, what is made of and what it can do.
- A class has:
  - fields: they represent the type of data of the class and the things it is made of;
  - methods: the actions that instances of that class can be asked to perform.
- An object is an instance of a class: X is an object of the class K if:
  - X provides memory to store data according to the structure described by K;
  - the methods described in K may be applied to the data in X.
- In other words, the class is a description of the storage layout and the functionality, while the object represents the corresponding data record.
- An object of a class, therefore, has its specific values of the data of the class and can be asked to perform the actions described be the methods of the class, in its specific way.
- The way it performs these actions might depend on the values of its data: a feature of object oriented programming is a strong binding between data and actions.
- In other words, a class is the blueprint of an object, while an object is a real instance of the class.

Example: the class `Human`

- A class `Human` might have the following *fields*:
  - `name`: once an object of this class is created, the programmer will give it its own value for the string `name`;
  - `age`: once an object of this class is created, the programmer will give it its own value for the integer `age`;
  - `nationality`: same thing;
  - two objects of the class `Leg`: they will help to perform the action `Walk`, see below;
  - and others.
- A class `Human` might have the following *methods*:
  - `walk`: performed with the help of the fields of type `Leg`;
  - `run`: same thing;
  - `speak`;
  - `eat`;
  - `sleep`;
  - and others
- Once you create the object `alessandroSgarabottolo` of the class `Human`, you can give it its own values for the fields of the class `Human` and ask it to perform some actions, i.e., call some methods.
- The way this object performs the method `run`, for example, might depend on the value of its age field: example of the binding between data and actions, i.e., between fields and methods.

More on the distinction between classes and objects: memory allocation

- The example "`alessandroSgarabottolo` is an object of the class `Human`" gives a very first intuition but does not tell the whole story.
- Something more can be said for example about the way data are stored: the class is the description of the memory layout needed to store objects of that class, whereas the object is a physical memory location that provides the memory.
- Indeed, the definition of a class exists only once, while the object of a class (data records) may exist multiple times.
- Example: if a class has a data field, and we create 100 objects (instances) of this class, then, of course, this consumes 100 times the memory of the data field. If a new method is added to a class, then its code is stored only once and the memory requirement is totally independent of the number of objects created.

Some more remarks about classes and objects

- Objects often interact with each other: think about an object of type *random variable* which gets summed with another random variable. Or maybe, two warriors fighting one against the other in a war videogame, or also a warrior destroying a house, again in a videogame. Both the warrior and the house are objects of some classes, `Warrior` and `House`, respectively.
- **In `Java`, everything is a class**: as soon as you want to write some code, that code must be written as part of a class.
- In this sense, here is a kind of disclaimer for our examples above: don't think about classes only as a way to represent something concrete, as a human, or a random variable, to come to something closer to our setting. You have to write a class to do anything, also testing the behavior of other classes, for example, or print a simple message (the first concrete example we will see).

Installing Java and Eclipse

Installing Java:

- Download and install the latest **Java Development Kit (JDK)** from Oracle or OpenJDK.
- Take care to download the version that is compatible with your machine.

What is Eclipse?
Eclipse is a popular integrated development environment (IDE) for Java that provides tools for coding, debugging, and project management. In particular in includes built-in support for Git and Maven.

Installing Eclipse:

- Download **Eclipse IDE for Java Developers** from eclipse.org.
- Run the installer and follow the setup instructions.
- Launch Eclipse and select a workspace (the folder where your projects will be stored).

Git and Maven

Git is a distributed version control system that allows developers to track changes in their code, collaborate efficiently, and manage project history.

- Distributed architecture for collaboration.
- Branching and merging capabilities.
- Tracks changes and enables version rollback.

Maven is a build automation and project management tool used primarily for Java projects. It helps manage dependencies, build processes, and project structure.

- Dependency management with a central repository.
- Standardized project structure using conventions.
- Automated builds with lifecycle management.

Integration with Eclipse:

- Eclipse provides built-in Git support through the **EGit** plugin, enabling version control operations like cloning, committing, branching, and pushing changes directly from the IDE.
- Eclipse includes **Maven Integration for Eclipse (m2e)**, allowing developers to manage dependencies, build projects, and execute Maven goals within the IDE.

Structure of a Maven Project in Eclipse

We create a Maven project choosing the option *Create a simple project (skip archetype selection)*

A Maven project in Eclipse follows a standard directory structure:

- **src/main/java** – contains the main Java source code.
- **src/main/resources** – stores configuration files and other resources.
- **src/test/java** – holds test classes for unit testing.
- **src/test/resources** – contains test-related configuration files.
- **pom.xml** – the Project Object Model (POM) file that manages dependencies and build configuration.
- **target/** – The output directory where compiled files and packaged artifacts are stored.

Eclipse automatically detects and manages Maven projects.

Note: if the project is not created correctly you can try skipping the option *Create a simple project* and select the archetype `maven-archetype-quickstart`.

Standard Naming Conventions in Java

It is recommended to follow a set of standard naming conventions to improve code readability and maintainability:

- **Classes and Interfaces**: use **PascalCase** (also known as UpperCamelCase)
  Example: `MyClass`, `RunnableInterface`
- **Methods and Variables**: use **camelCase** (lowercase first letter)
  Example: `calculateTotal()`, `userAge`
- **Packages**: Use all lowercase, typically following a reverse domain naming convention
  Example: `com.alessandrosgarabottolo.session1`

Hello World! and the `main` Method

The `main` method is the entry point of a Java application. It has a specific signature:

```
public static void main(String[] args)
```

Explanation:

- `public` - The method is accessible from anywhere.
- `static` - It can be run without creating an instance of the class.
- `void` - It does not return a value.
- `main` - The specific method name required by Java.
- `String[] args` - Accepts command-line arguments as an array of Strings.

Example:

```
public class HelloWorld {
 public static void main(String[] args) {
  System.out.println("Hello, World!");
 }
}
```

Cf. `com.alessandrosgarabottolo.session1.helloworld.HelloWorld`

Comments and Documentation

Java provides different ways to write comments and documentation:
Single-line comments: use '//' to write a brief comment on a single line.

```
// This is a single-line comment
int x = 10; // Variable declaration
```

Multi-line comments: use '/* ... */' for block comments.

```
/* This is a multi-line comment
used for longer explanations */
```

Javadoc comments: use '/** ... */' for generating documentation.

```
/**
* Computes the sum of two numbers.
* @param a First number
* @param b Second number
* @return Sum of a and b
*/
```

Object Initialization

Objects are (almost always) initialized using the syntax

```
NameOfClass nameOfObject = new NameOfClass()
```

See `com.alessandrosgarabottolo.oophelloworld`. Here we see how methods are defined and implemented, and how fileds are initialized and used by methods.

Primitives

In Java, primitive types are the most basic data types and are not objects. They represent simple values and are used to perform operations directly on the data. In fact, primitives are stored as values (in the stack), while objects are stored as references (more on this later).
The primitive types in Java are:

- **byte**: 8-bit signed integer
- **short**: 16-bit signed integer
- **int**: 32-bit signed integer, range $[-2^{31}, 2^{31} - 1]$
- **long**: 64-bit signed integer, range $[-2^{63}, 2^{63} - 1]$
- **float**: 32-bit floating-point number, precision ~ 7 decimal digits
- **double**: 64-bit floating-point number, precision ~ 16 decimal digits
- **char**: 16-bit Unicode character
- **boolean**: Represents `true` or `false`

Java provides **wrapper classes** that allow you to treat primitives as objects when needed. The respective wrappers for the previous primitives are **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, **Character**, **Boolean**.

For operations on primitives see `com.alessandrosgarabottolo.operators.Operators`.

Assigning and referencing objects

Pay attention to how you treat variables. if a variable is an instance of a class, it is actually a reference to the location in the memory where the object is stored.

See `com.alessandrosgarabottolo.session1.tank`

if...else statement and ternary operator

if...else statement:

```
if (condition) {
  // code if condition is true
}
else {
  // code if condition is false
 }
```

Ternary Operator: a shorthand for an if-else statement with a single expression.

```
condition ? expression1 : expression2;
```

If the condition is true, `expression1` is evaluated; otherwise, `expression2`.

See

- `com.alessandrosgarabottolo.session1.divisible.Divisible`
- `com.alessandrosgarabottolo.session1.testval`

do...while Statement and for loop

The do...while loop executes a block of code at least once, then repeatedly as long as a condition is true.

Syntax:

```
do {
 // code to execute
} while (condition);
```

- The loop body executes at least once, regardless of the condition.
- The condition is checked after executing the loop body.

See com.alessandrosgarabottolo.session1.randomvariable.WhileRandom.

The for loop is a control statement that executes a block of code a fixed number of times.

Syntax:

```
for (initialization; condition; update) {
 // code to execute
}
```

See

- com.alessandrosgarabottolo.session1.elevator.Elevator
- com.alessandrosgarabottolo.session1.gauss
- com.alessandrosgarabottolo.session1.primenumbers

switch statement

The switch statement is a control statement that allows the execution of different code blocks based on the value of an expression.

Syntax:

```
switch (expression) {
 case value1:
 // code block
 break;
 case value2:
 // code block
 break;
 ...
 default:
 // default code block
}
```

- The expression must evaluate to a value of type int, char, enum, String, or certain wrapper types.
- Each case checks for a specific value.
- The break statement prevents fall-through to the next case.
- The default case is optional and executes if no other case matches.

See com.alessandrosgarabottolo.session1.switches.ARandomSwitch