A brief introduction to the main ideas of Object Oriented Programming

Andrea Mazzon

Ludwig Maximilians Universität München

Machine model vs Model of the problem

- Suppose you want to solve a problem by using your computer.
- The first effort you have to make 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.
- That is, you want to represent your problem the closest possible to the way you think about the problem.
- 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.
- The effort required to perform this mapping produces programs that are difficult to write and expensive to maintain.
- The effort to provide 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 as 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 the user to program in a similar way as she/he works with real-life entities.
- The sub-elements are represented by objects.
- The idea is that the program is allowed to adapt itself 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 `AndreaMazzon` 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 "`AndreaMazzon` 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 by 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).