Object Oriented Programming

Prof. Michele Loreti

Programmazione Avanzata
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

Working with Objects

In functional programming programs mainly rely on function calls.

Working with Objects

In functional programming programs mainly rely on function calls.

Functions can be considered as black boxes:

Working with Objects

In functional programming programs mainly rely on function calls.

Functions can be considered as black boxes:

- when we invoke a function we are interested in the result;

Working with Objects

In functional programming programs mainly rely on function calls.

Functions can be considered as black boxes:

- when we invoke a function we are interested in the result;
- data are separated from operations.

Working with Objects

In functional programming programs mainly rely on function calls.

Functions can be considered as black boxes:

- when we invoke a function we are interested in the result;
- data are separated from operations.

Working with Objects

In functional programming programs mainly rely on function calls.

Functions can be considered as black boxes:

- when we invoke a function we are interested in the result;
- data are separated from operations.

In Object Oriented programming you have another dimension:

Working with Objects

In functional programming programs mainly rely on function calls.

Functions can be considered as black boxes:

- when we invoke a function we are interested in the result;
- data are separated from operations.

In Object Oriented programming you have another dimension:

- each object can have its own state;

Working with Objects

In functional programming programs mainly rely on function calls.

Functions can be considered as black boxes:

- when we invoke a function we are interested in the result;
- data are separated from operations.

In Object Oriented programming you have another dimension:

- each object can have its own state;
- the state effect the result you get from calling a method.

Working with Objects

In functional programming programs mainly rely on function calls.

Functions can be considered as black boxes:

- when we invoke a function we are interested in the result;
- data are separated from operations.

In Object Oriented programming you have another dimension:

- each object can have its own state;
- the state effect the result you get from calling a method.

Working with Objects

In functional programming programs mainly rely on function calls.

Functions can be considered as black boxes:

- when we invoke a function we are interested in the result;
- data are separated from operations.

In Object Oriented programming you have another dimension:

- each object can have its own state;
- the state effect the result you get from calling a method.

Example: Let `in` be a `Scanner` object, if we call `in.next()` the object remembers what was read before and gives us the next token.

Working with Objects

When you use objects (that someone else implemented) and invoke methods on them, you do not need to know what does under the hood.

Working with Objects

When you use objects (that someone else implemented) and invoke methods on them, you do not need to know what does under the hood.

This mechanism is named encapsulation. This is a key concept in object oriented programming.

Working with Objects

When you use objects (that someone else implemented) and invoke methods on them, you do not need to know what does under the hood.

This mechanism is named encapsulation. This is a key concept in object oriented programming.

If we want to make available your code to other developers, we have to make available your objects via classes.

Example: Managing Calendars

Managing calendars is a common tasks. However it is not an easy work since you have to manage:

- varying of months lenght;
- leap years;
- leap seconds!

Example: Managing Calendars

Managing calendars is a common tasks. However it is not an easy work since you have to manage:

- varying of months lenght;
- leap years;
- leap seconds!

Expert in the field can provide the classes that provides expected features:

- a class for managing the concept of date;
- implementing date arithmetics.

Example: Calendar application in Java

Implement an application that mimic the Unix `cal` program.

Example: Calendar application in Java

Implement an application that mimic the Unix cal program.

```
Micheles-MBP:~ loreti$ cal

     March 2018
Su Mo Tu We Th Fr Sa
             1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

Example: Calendar application in Java

Question: How can we implement such program?

Example: Calendar application in Java

Question: How can we implement such program?

We can first use the LocalDate class to express a date at some unspecified location.

Example: Calendar application in Java

Question: How can we implement such program?

We can first use the LocalDate class to express a date at some unspecified location.

We need an object of that class that represents the first day of the month:

```
LocalDate date = LocalDate.now().withDayOfMonth(1);
```

Example: Calendar application in Java

Question: How can we implement such program?

We can first use the LocalDate class to express a date at some unspecified location.

We need an object of that class that represents the first day of the month:

```
LocalDate date = LocalDate.now().withDayOfMonth(1);
```

By invoking method date.plusDays(1) you can advance the date by 1 day. The result is a new LocalDate object:

```
    date = date.plusDays(1);
```

Example: Calendar application in Java

We can use this information to print the calendar:

```java
int counter = 1;
while (date.getMonthValue() == 3) {
  if (counter == 8) {
    System.out.println();
    counter = 1;
  }
  System.out.printf("%4d",date.getDayOfMonth());
  date = date.plusDays(1);
  counter++;
}
```

Example: Calendar application in Java

Method getDayOfWeek() can be used to get weekday on which the date fall:

```
DayOfWeek weekday = date.getDayOfWeek();
```

Example: Calendar application in Java

Method getDayOfWeek() can be used to get weekday on which the date fall:

```
DayOfWeek weekday = date.getDayOfWeek();
```

We can get numerical value of weekday to compute the correct indentation of the first day in the month:

```
int value = weekday.getValue();
for (int i=1; i<value ; i++) {
  System.out.print("     ");
}
```

Example: Calendar application in Java

```java
public static void main(String[] argv) {
  LocalDate date = LocalDate.now().withDayOfMonth(1);
  int month = date.getMonthValue();
  System.out.println(" Mon Tue Wed Thu Fri Sat Sun");
  DayOfWeek weekday = date.getDayOfWeek();
  int value = weekday.getValue();
  for (int i=1; i<value ; i++) {
    System.out.print("    ");
  }
  while (date.getMonthValue() == month) {
    System.out.printf("%4d",date.getDayOfMonth());
    date = date.plusDays(1);
    if (date.getDayOfWeek().getValue()==1) {
      System.out.println();
    }
  }
}
```

Accessor and Mutator Methods

We have two kinds of methods:

- accessors that are used to retrieve info from an object:

```
date.plusDays(1)
```

- mutators that change the state of the object in which it was invoked.

Accessor and Mutator Methods

We have two kinds of methods:

- accessors that are used to retrieve info from an object:

```
date.plusDays(1)
```

- mutators that change the state of the object in which it was invoked.

All methods of class LocalDate are accessors!

Accessor and Mutator Methods

We have two kinds of methods:

- accessors that are used to retrieve info from an object:

```
    date.plusDays(1)
```

- mutators that change the state of the object in which it was invoked.

All methods of class LocalDate are accessors!

An example of mutator method is:

```
ArrayList<String> beverages = new ArrayList<>();
beverages.add("Beer");
```

Object References

In Java a variable can only holds references to an object.

Object References

In Java a variable can only holds references to an object.

If we assign a variable holding an object to another variable, we have two references to the same object:

```
ArrayList<String> drinks = beverages;
```

Object References

In Java a variable can only holds references to an object.

If we assign a variable holding an object to another variable, we have two references to the same object:

```
ArrayList<String> drinks = beverages;
```

When we change the object, the mutation is observed by both the references:

```
drinks.add("Cola"); //The size of beverages is 2!
```

Sharing an object is efficient and convenient! But it could be dangerous!

Implementing Classes

We consider a standard example in object oriented: the class of employees.

Implementing Classes

We consider a standard example in object oriented: the class of employees.

An employee has:

- a name;
- a salary.

Implementing Classes

We consider a standard example in object oriented: the class of employees.

An employee has:

- a name;
- a salary.

Name and salary are the values in the state of an employee object. In Java these are rendered as instance variables:

```
public class Employee {
  private String name;
  private double salary;
  ...
}
```

Method Headers

We can now implement methods for the Employee.

Method Headers

We can now implement methods for the Employee.

When we declare a method we provide:

Method Headers

We can now implement methods for the Employee.

When we declare a method we provide:

- a name;

Method Headers

We can now implement methods for the Employee.

When we declare a method we provide:

- a name;
- types and names of its parameters;

Method Headers

We can now implement methods for the Employee.

When we declare a method we provide:

- a name;
- types and names of its parameters;
- return type.

Method Headers

We can now implement methods for the Employee.

When we declare a method we provide:

- a name;
- types and names of its parameters;
- return type.

Method Headers

We can now implement methods for the Employee.

When we declare a method we provide:

- a name;
- types and names of its parameters;
- return type.

For instance:

```
public void raiseSalary(double byPercent) {
   ...
}

public String getName() {
   ...
}
```

Method Bodies

We have to define a body for our methods:

```java
public void raiseSalary( double byPercent ) {
  double raise = this.salary*byPercent/100;
  this.salary += raise;
}

public void getName() {
  return this.name;
}
```

Method Bodies

We have to define a body for our methods:

```
public void raiseSalary( double byPercent ) {
  double raise = this.salary*byPercent/100;
  this.salary += raise;
}

public void getName() {
  return this.name;
}
```

The keyword this is used to refer to the object that received the invocation of the method.

Object construction

The last step to complete our Employee is to provide a constructor.

Object construction

The last step to complete our Employee is to provide a constructor.

A constructor is similar to declaring a method. However:

- the name of the constructor must be the same as the class name;
- there is no return type.

Object construction

The last step to complete our Employee is to provide a constructor.

A constructor is similar to declaring a method. However:

- the name of the constructor must be the same as the class name;
- there is no return type.

```
public Employee( String name , double salary ) {
  this.name = name;
  this.salary = salary;
}
```

Object construction

The last step to complete our Employee is to provide a constructor.

A constructor is similar to declaring a method. However:

- the name of the constructor must be the same as the class name;
- there is no return type.

```
public Employee( String name , double salary ) {
  this.name = name;
  this.salary = salary;
}
```

A constructor executes when we use the new operator:

```
new Employee("Peter Parker",1000);
```

Overloading

We can have more than one version of the constructor:

```
public Employee( double salary ) {
  this.name = ””;
  this.salary = salary;
}
```

Overloading

We can have more than one version of the constructor:

```
public Employee( double salary ) {
  this.name = "";
  this.salary = salary;
}
```

Our class has now two constructors, and we say that the constructor is overloaded.

Overloading

We can have more than one version of the constructor:

```
public Employee( double salary ) {
  this.name = "";
  this.salary = salary;
}
```

Our class has now two constructors, and we say that the constructor is overloaded.

To avoid duplicated code, we can call one constructor from the other:

```
public Employee( double salary ) {
  this("",salary);
}
```

Default initialisation

If a field is not assigned in a constructor, it is automatically assigned to a default value:

- 0 for numerical values;
- false for booleans;
- null for objects.

Default initialisation

If a field is not assigned in a constructor, it is automatically assigned to a default value:

- 0 for numerical values;
- false for booleans;
- null for objects.

```
public Employee( String name ) {
  this.name = name;
  //Salary is automatically set to zero!
}
```

It is convenient, to avoid errors, to explicitly assign all fields that are objects!

Instance Variable Initialisation

Instance variable can have a default initial value:

```
public class Employee {

  private String name = "";
```

Instance Variable Initialisation

Instance variable can have a default initial value:

```java
public class Employee {

  private String name = "";
```

The initialisation occur after an object has been allocated and before a constructor runs.

Instance Variable Initialisation

Instance variable can have a default initial value:

```
public class Employee {

  private String name = "";
```

The initialisation occur after an object has been allocated and before a constructor runs.

Constructors may overwrite this value!

Initialisation blocks

We can include arbitrary initialisation blocks:

Initialisation blocks

We can include arbitrary initialisation blocks:

```
public class Employee {

  private String name = "";
  private double salary;
  private int id;

  {
    Random generator = new Random();
    id = 1+generator.nextInt(1_000_000);
  }
```

Final Instance Variables

We can declare an instance variable final . In this case this must be initialised at the end of any constructor.

Final Instance Variables

We can declare an instance variable `final`. In this case this must be initialised at the end of any constructor.

Afterwards, the variable may not be modified again.

Final Instance Variables

We can declare an instance variable final . In this case this must be initialised at the end of any constructor.

Afterwards, the variable may not be modified again.

```
public class Employee {
  private final String name;
  ....
}
```

Default constructor

A special constructor is the one without arguments:

Default constructor

A special constructor is the one without arguments:

```
public Employee( ) {
  this.name = "";
  this.salary = 0;
}
```

Default constructor

A special constructor is the one without arguments:

```
public Employee( ) {
  this.name = "";
  this.salary = 0;
}
```

A class with no constructors is automatically equipped with a default constructor.

Static variables

We can declare a variable as static . This is associated with the class and shared among all the instances.

```
public class Employee {
  private static int lastId = 0;
  private int id;

  public Employee() {
    lastId++;
    id = lastId;
  }
  ...
}
```

Static variables

We can declare a variable as static. This is associated with the class and shared among all the instances.

```
public class Employee {
  private static int lastId = 0;
  private int id;

  public Employee() {
    lastId++;
    id = lastId;
  }
  ...
}
```

Mutable static variables should be used with attention. However, constants are quite common:

```
public static final duble PI = 3.1415...
```

Static initialisation blocks

A `static` initialisation block can be used to initialise code at the level of classes.

Static initialisation blocks

A `static` initialisation block can be used to initialise code at the level of classes.

These are used when we need to perform complex computations to initialise static variables or constants.

Static initialisation blocks

A static initialisation block can be used to initialise code at the level of classes.
These are used when we need to perform complex computations to initialise static variables or constants.

```
public class CreditCardForm {

  private static final ArrayList<Integer> expirationYear =
    new ArrayList<>();

  static {
    int year = LocalDate.now().getYear();
    for( int i=year; i<year+20; i++ ) {
      expirationYear.add(year);
    }
  }

  ...

}
```

Static Methods

Static methods are methods that do not operate on objects:

```
Math.pow(x, a);
```

Static Methods

Static methods are methods that do not operate on objects:

```
Math.pow(x, a);
```

A common use of `static` methods is for factory methods.

Static Methods

Static methods are methods that do not operate on objects:

```
Math.pow(x, a);
```

A common use of `static` methods is for factory methods.

These are used to build an object.

Static Methods

Static methods are methods that do not operate on objects:

```
Math.pow(x,a);
```

A common use of `static` methods is for factory methods.

These are used to build an object.

Question: Why not use a constructor?

Static Methods

Static methods are methods that do not operate on objects:

```
Math.pow(x, a);
```

A common use of static methods is for factory methods.

These are used to build an object.

Question: Why not use a constructor?

- we can built different objects with the same parameters:

```
NumberFormat.getCurrencyInstance()
NumberFormat.getPercentInstance()
```

Static Methods

Static methods are methods that do not operate on objects:

```
Math.pow(x, a);
```

A common use of static methods is for factory methods.

These are used to build an object.

Question: Why not use a constructor?

- we can built different objects with the same parameters:

```
NumberFormat.getCurrencyInstance()
NumberFormat.getPercentInstance()
```

- we can obtain instances of a subclass

Static Methods

Static methods are methods that do not operate on objects:

```
Math.pow(x, a);
```

A common use of `static` methods is for factory methods.

These are used to build an object.

Question: Why not use a constructor?

- we can built different objects with the same parameters:

```
NumberFormat.getCurrencyInstance()
NumberFormat.getPercentInstance()
```

- we can obtain instances of a subclass
- we are independent from a specific implementation!

Packages. . .

In `Java` classes are placed into packages.

Packages. . .

In Java classes are placed into packages.

We can organise our code that can be structured according the use:

- java . lang
- java . util
- java . math
- . . .

Packages. . .

In Java classes are placed into packages.

We can organise our code that can be structured according the use:

- java . lang
- java . util
- java . math
- . . .

Packages guarantee the uniqueness of class name!

Package declaration

A package name is a dot-separated list of identifiers

```
java.util.regex
```

Package declaration

A package name is a dot-separated list of identifiers

```
java.util.regex
```

To guarantee unique package names it is a good idea to use an Internet domain name (written in the reverse order):

`http://quasylab.unicam.it` ⟶ it.unicam.quasylab
`http://quanticol.github.io` ⟶ io.quanticol.github
`http://pspaces.github.io` ⟶ io.quanticol.pspaces

Java packages do not nest: there is no relation between `java.util` and `java.util.regex`.

Package declaration

```
package it.unicam.cs.pa;

public class Employee {
  ...
}
```

Package declaration

```java
package it.unicam.cs.pa;

public class Employee {
   ...
}
```

Each class has a fully qualified name:

packagename.ClassName

Package declaration

```
package it.unicam.cs.pa;

public class Employee {
   ...
}
```

Each class has a fully qualified name:

packagename.ClassName

There exists also a default package that contains all classes without a package declaration. It is use is not recommended!

Package declaration

```
package it.unicam.cs.pa;

public class Employee {
   ...
}
```

Each class has a fully qualified name:

packagename.ClassName

There exists also a default package that contains all classes without a package declaration. It is use is not recommended!

The path of a class must match the structure of the file system:

$$\text{it.unicam.cs.pa} \longrightarrow \texttt{it/unicam/cs/ps}$$

Compiling a Java class

Each Java projects should be structured with the following folders:

- src: that contains all source files;
- bin: where the .class files are generated;
- libs: with the required libraries;
- doc: with all the documentation.

Compiling a Java class

Each Java projects should be structured with the following folders:

- src: that contains all source files;
- bin: where the .class files are generated;
- libs: with the required libraries;
- doc: with all the documentation.

To build a .java file the following command (executed in the src folder) should be used:

```
javac packagepath/Classname.java
```

Compiling a Java class

Each Java projects should be structured with the following folders:

- src: that contains all source files;
- bin: where the .class files are generated;
- libs: with the required libraries;
- doc: with all the documentation.

To build a .java file the following command (executed in the src folder) should be used:

```
javac packagepath/Classname.java
```

A tool supporting the building of Java projects is crucial!

Classpath

A class path is a path with all the dependencies needed by our code.

Classpath

A class path is a path with all the dependencies needed by our code.

In the class path we can include:

- Directories containing class (in the appropriate subdirectories);
- JAR files;
- Directories containing JARs.

Classpath

A class path is a path with all the dependencies needed by our code.

In the class path we can include:

- Directories containing class (in the appropriate subdirectories);
- JAR files;
- Directories containing JARs.

The class path can be passed to the compiler after the parameter -cp:

```
javac -cp .:../libs/\* packagepath/Classname.java
```

Package Access

We have already seen the modifiers `public` and `private`.

Package Access

We have already seen the modifiers `public` and `private` .

A `public` feature can be accessed by any class.

Package Access

We have already seen the modifiers `public` and `private`.

A `public` feature can be accessed by any class.

A `private` feature can be accessed only by class that declare it.

Package Access

We have already seen the modifiers `public` and `private` .

A `public` feature can be accessed by any class.

A `private` feature can be accessed only by class that declare it.

If a feature has not any modifier its visibility is at the level of package: all classes in the same package can use that feature!

Package Access

We have already seen the modifiers `public` and `private`.

A `public` feature can be accessed by any class.

A `private` feature can be accessed only by class that declare it.

If a feature has not any modifier its visibility is at the level of package: all classes in the same package can use that feature!

By default any package is open ended: new classes can be added to a package!

Importing classes

The import statement can be used to import classes that can be used without the fully qualified name:

```
import java.util.Ramdom;
```

Importing classes

The import statement can be used to import classes that can be used without the fully qualified name:

```
import java.util.Ramdom;
```

You can import all classes of a package by using the symbol *:

```
import java.util.*;
```

This approach is discouraged!

Importing classes

The import statement can be used to import classes that can be used without the fully qualified name:

```
import java.util.Ramdom;
```

You can import all classes of a package by using the symbol ∗:

```
import java.util.*;
```

This approach is discouraged!

Static import can be used to import all (or specific) static methods and variables defined in a class:

```
import static java.lang.Math.*;
```

Importing classes

The import statement can be used to import classes that can be used without the fully qualified name:

```
import java.util.Ramdom;
```

You can import all classes of a package by using the symbol ∗:

```
import java.util.*;
```

This approach is discouraged!

Static import can be used to import all (or specific) static methods and variables defined in a class:

```
import static java.lang.Math.*;
```

After that you can use all the static methods in Math without prefix.

To be continued. . .