Unit 8

Inheritance

Summary

- Inheritance
- Overriding of methods and polymorphism
- The class `Object`

8.1 Inheritance

Inheritance in object-oriented languages consists in the possibility of defining a class that is the *specialization* of an existing class: i.e., defining a class that has the same properties as an already existing class, but to which we add new functionalities and/or new information.

Instead of modifying the already defined class, we create a new class *derived* from it.

For example, suppose we have defined a class `Person`:

```
public class Person {
  ...
}
```

We can derive the class `Student` from `Person`:

```
public class Student extends Person {
  ...
}
```

We say that:

- `Student` is a *subclass* of `Person`. `Person` is a *superclass* of `Student`.
- `Student` is a *class derived* from the *base class* `Person`.

A subclass *inherits* all the methods and all the instance variables of the superclass, and additionally it can have its own methods and instance variables.

8.2 Inheritance: example

Suppose we have defined a class `Person` as follows:

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

  public Person(String n, String r) {   // constructor
    name = n;     residence = r;
  }
  public String getName() {
    return name;
  }
  public String getResidence() {
    return residence;
  }
  public void setResidence(String newResidence) {
```

```
    residence = newResidence;
  }
}
```

We derive the subclass `Student` from the class `Person` as follows:

```
public class Student extends Person {
  private String faculty;

  public Student(...) {   // constructor
    ...
  }

  public String getFaculty() {
    return faculty;
  }
}
```

The objects of the class `Student` are characterized by the properties *inherited* from the class `Person` and *additionally* by the faculty at which the student is registered.

8.3 Fundamental features of derived classes

- All the properties (instance variables and methods) defined for the base class are implicitly defined also for the derived class, i.e., they are inherited by the derived class.
- The derived class can have additional properties with respect to those inherited from the base class.
- Each instance of the derived class is also an instance of the base class. Hence, it is possible to use an object of the derived class in each situation in which it is possible to use an object of the base class.
- Note that the converse is not true, i.e., it is *not* possible to use an object of the base class in each situation in which it is possible to use an object of the derived class (see later).

8.4 Constructor of a derived class

Let us analyze now how to define constructors in the presence of derivation between classes. Note that a constructor of a derived class must also take care of the construction of the fields of the base class. This can be done by inserting in the constructor of the derived class the call to a constructor of the base class, using the special Java construct `super()`. The `super()` statement must appear as the *first executable statement* in the body of the constructor of the derived class. For example:

```
public class Student extends Person {
  public Student(String n, String r, String f) {
    super(n,r);  // calls the constructor Person(String,String)
    faculty = f;
  }
  ...
}
```

The statement `super(n,r);` calls `Person(n,r)`, which initializes the instance variables `name` and `residence`, inherited from the superclass `Person`, respectively to the strings `n` and `r`. Then, the statement `faculty = f;` assigns to the instance variable `faculty` the string `f`.

8.5 Use of `super()`

In general, when the subclass has its own instance variables, its constructor must first construct an object of the superclass (using `super()`) and then construct its own instance variables.

Note:

- What happens if we forget to insert `super()`? Then the constructor with no arguments of the superclass is called automatically (obviously, if the constructor with no arguments is not defined for the superclass, then a compilation error is signaled).

- What happens if we forget to define constructors for the subclass? Then a constructor with no arguments is automatically defined; such a constructor calls the constructor with no arguments of the superclass and initializes the proper (non inherited) instance variables of the subclass to the default values.
- In this course, we will never make use of these automatic definitions. Instead, we will always explicitly define the constructors of a subclass in such a way that in their first statement they call `super()`.

8.6 Inherited methods and variables

From what we have said, all the objects of the class `Student`, besides having the proper methods and instance variables defined in `Student`, *inherit* all methods and instance variables of `Person`. For example, we can write:

```
public class TestStudent {
  public static void main(String[] args) {
    Person p = new Person("Daniele", "Roma");
    System.out.println(p.getName());
    System.out.println(p.getResidence());
    Student s = new Student("Jacopo", "Roma", "Engineering");
    System.out.println(s.getName());       // OK! method inherited from Person
    System.out.println(s.getResidence());  // OK! method inherited from Person
    System.out.println(s.getFaculty());    // OK! method defined in Student
  }
}
```

The methods `getName()` and `getResidence()`, inherited from `Person` are effectively methods of the class `Student`.

8.7 Compatibility

We have said that each object of the derived class is also an object of the base class. This implies that it is possible to use an object of the derived class in each situation or context in which it is possible to use an object of the base class. In other words, *the objects of the derived class are* ***compatible*** *with the objects of the base class.*

But the contrary is not true! Consider the following program.

```
public class TestCompatibility {
  public static void main(String[] args) {
    Person p = new Person("Daniele", "Roma");
    Student s = new Student("Jacopo", "Roma", "Engineering");
    Person pp;
    Student ss;
    pp = s;    //OK! Student is compatible with Person
    ss = p;    //ERROR! Person is not compatible with Student
    System.out.println(pp.getName());
               //OK! getName() is a method of Person
    System.out.println(pp.getResidence());
               //OK! getResidence is a method of Person
    System.out.println(pp.getFaculty());
               //ERROR! getFaculty is not a method of Person
  }
}
```

Note: The error in the last statement is due to the fact that the variable `pp` is a reference to a `Person`, and hence, through this variable it is not possible to access the methods of `Student` (even if in this case `pp` actually refers to an object `Student`). This is because Java implements *static type checking.*

8.8 Compatibility between actual and formal parameters

What we have seen regarding compatibility between superclass and subclass applies also to parameter passing.

```
public class TestCompatibility2 {
  public static void printPerson(Person p) {
    System.out.println(p.getName());
    System.out.println(p.getResidence());
  }

  public static void printStudent(Student s) {
    System.out.println(s.getName());
    System.out.println(s.getResidence());
    System.out.println(s.getFaculty());
  }

  public static void main(String args[]) {
    Person pr = new Person("Daniele", "Roma");
    Student st = new Student("Jacopo", "Roma", "Engineering");
    printPerson(pr);   //OK
    printPerson(st);   //OK! Student is compatible with Person
    printStudent(st);  //OK
    printStudent(pr);  //ERROR! Person is not compatible with Student
  }
}
```

8.9 Access to the public and private fields of the superclass

As we have seen, the derived class inherits all instance variables and all methods of the superclass.

Obviously, the public fields of the superclass are accessible from the derived class. For example, we can add to the subclass `Student` a method `printName()` as follows.

```
public class Student extends Person {
  ...
  public void printName() {
    System.out.println(this.getName());
  }
  ...
}
```

What about the private fields of the superclass? More precisely, are the methods defined in the derived class considered as any other client of the superclass, or do they have special privileges to access the private fields of the superclass? The answer is that *the private fields of the superclass are* ***not accessible*** *to the methods of the derived class*, exactly as they are not accessible to any other method outside the superclass.

For example, if we introduce in `Student` a method `changeName()` as follows, we get a compilation error.

```
public class Student extends Person {
  ...
  public void changeName(String s) {
    this.name = s; //ERROR! the instance variable name is private in Person
                   //hence, it is not accessible from the derived class Student
  }
  ...
}
```

Note: Java allows us to use, besides public and private fields, also fields of a different type, called `protected`. The protected fields of a class cannot be accessed by external methods, but they can be accessed by the methods of a derived class. We will not make use of protected fields in this course.

8.10 Overriding of methods

- We say that we do **overriding** of a method `m()` when we define in the subclass a method `m()` having *exactly the same signature* as the method `m()` in the superclass.

- When we do overriding, Java requires that the definition of the new method `m()` has also the same return type as the original method `m()`. In other words, the method we are redefining must have the *same header* as the original method.
- The consequence of overriding is that, each time we invoke the method `m()` on an object of the derived class `D`, the method that is effectively called is the one redefined in `D`, and not the one defined in the base class `B`, even if the reference used to denote the invocation object is of type `B`. This behavior is called *polymorphism*.
- Note that *overriding* is different from *overloading* (which is the definition of two methods with the same name but different signatures).

8.11 Overriding of methods: example

Assume that in `Person` we define a method `printData()` as follows:

```
public class Person {
  ...
  public void printData() {
    System.out.println(name + "  " + residence);
  }
  ...
}
```

We do overriding of the method `printData()` in the class `Student`, in such a way that `printData()` prints also the faculty:

```
public class Student extends Person {
  ...
  public void printData() { // overriding of printData of Person!!!
    System.out.println(this.getName() + "  " + this.getResidence() + "  "
                       + faculty);
  }
  ...
}
```

An example of client is the following.

```
public class ClientStudent {
   public static void main(String[] args) {
     Person p = new Person("Daniele", "Roma");
     Student s = new Student("Jacopo", "Roma", "Engineering");
     p.printData();
     s.printData();
   }
}
```

8.12 Polymorphism

The overriding of methods causes **polymorphism**, which means the presence in a class hierarchy of methods with the same signature that behave differently.

Consider the following program.

```
public class StudentPolymorphism {
  public static void main(String[] args) {
    Person p = new Person("Daniele", "Roma");
    Student s = new Student("Jacopo", "Roma", "Engineering");
    Person ps = s; // OK! due to the compatibility rules
    p.printData();
    s.printData();
    ps.printData();    // ??? what does this print ???
  }
```

```
}
```

The method `printData()` which is effectively called is chosen based on the class the object belongs to, and not based on the type of the variable denoting it. This mechanism for accessing methods is called *late binding*.

In the above example, the method called on the object `ps` will be the one defined in the class `Student`, i.e., the one that prints name, residence, and faculty. In fact, if we execute the program, it will print:

```
Daniele  Roma
Jacopo  Roma  Engineering
Jacopo  Roma  Engineering
```

8.13 Class hierarchy

- A class can have several subclasses. For example, we could define a subclass `ExpertPerson` of `Person`, whose objects represent persons that are experts in a certain topic, where the topic in which they are experts is a specific property of the class.
- A subclass of a class can itself have subclasses. For example, the class `Student` derived from `Person` could have a subclass `WorkingStudent`.
- Hence, using several derivations, it is possible to create hierarchies of classes.

8.14 The class `Object`

All classes defined in Java are subclasses of the predefined class `Object`, even if this is not indicated explicitly.

This means that all classes inherit from `Object` several standard methods, such as `equals()`, `clone()`, and `toString()`, which are defined for `Object`. If we invoke one of these methods on some object belonging to a class `C`, but have not overridden it in `C` (or in one of its superclasses different from `Object`), the method defined in the class `Object` is used.

Now we can understand better that the possibility of "redefining" the `toString()` method discussed in Unit 3, actually corresponds to overriding the one of the class `Object`.

Note: The `toString()` method is automatically invoked by the `print()` and `println()` methods. This works correctly because of the mechanism of *late-binding*.

8.15 Composition (optional)

Let us define a class `Student2` with functionalities analogous to `Student`, but which does not make use of inheritance. The idea is to include in `Student2` an instance variable that is a reference to an object `Person`. Such an instance variable is used to maintain the name and residence properties, and we add to it an instance variable `faculty`, used to store the faculty.

```
public class Student2 {
  private Person person;
  private String faculty;

  public Student2(String name, String residence, String faculty) {
    person = new Person(name, residence);
    this.faculty = faculty;
  }

  public String getName() {
    return person.getName();
  }

  public String getResidence() {
    return person.getResidence();
  }

  public void setResidence(String residence) {
    person.setResidence(residence);
```

```
    }

    public String getFaculty() {
      return faculty;
    }
  }
```

Note:

- The class `Student2` uses an instance variable `Person`; the object `Person` stores name and residence of the student. Note that `Student2` is a client of `Person`, hence the field `Person` is manipulated using the public methods of the class `Person`.
- The class `Student2` hides completely from its clients the use of the object `Person`. Indeed, it offers its clients the methods `getName()`, `getResidence()`, and `setResidence()`, that operate on `Student2` objects.
- The class `Student2` offers its clients the same operations (methods) as the class `Student`. However, the objects `Student2` *are not compatible* with the objects of the class `Person`. Hence, if a variable or a formal parameter is of type (reference to an object) `Person`, it cannot contain (a reference to) an object `Student2`.

8.16 Inheritance or composition? (optional)

Referring to the previous example, it is clear that realizing the class `Student2` instead of `Student` is a questionable choice. So, when is it reasonable to use composition?

In general, we can adopt the following criteria:

- If each object of X is an object of Y (X IS-A Y), then we use inheritance.
- If each object of Y has an object of X (Y HAS-A X), then we use composition.

These aspects will be studied in later courses.

Exercises

Exercise 8.1. Extend the class `Book` (Exercise 3.1), by defining other classes:

- for textbooks, where we can specify also the course to which they refer;
- for textbooks of the Free University of Bolzano, where we can specify also the faculty for which the book has been chosen;
- for novels, where we can specify also the type.

For each of the above classes, define the constructor and suitably redefine the method for the visualization of the object. Describe the hierarchy of classes.

Exercise 8.2. Write an example program for the classes defined in Exercise 8.1, that realizes the following operations.

1. for a first book, read the title, the authors, the course, and the faculty adopting it, and store this information in an object;
2. show the information about the first book;
3. for a second book, read the title and the authors, and store them in a suitable object;
4. read the topic of the second book, and create a third object of type novel;
5. show the information about the two objects used to create the second book;
6. read the price of the first book and update the corresponding object accordingly;
7. show the information about the first book, including the price.