Chapter 8 Exercises

Polimorphism

In this chapter we will simulate the construction of an IDE, in a very simplified way. Also in this case, it will be an incremental exercise so you have to complete each exercise (and read the solution) in order to move on to the next one.
We will create classes and interfaces step by step. In particular we will create the abstractions of `IDE`, `Editor`, `SourceFile`, `File`, `FileType` and so on. The exercise is guided, so the reader is relieved of the responsibility to decide which classes should compose our application. For this reason, there are exercises on programming rather than analysis and design.
With the next exercises we will only have to apply the definitions we have learned so far, almost no algorithm is required. The ultimate goal is to focus on the Object Orientation and not on the algorithms. In addition, other types of exercises are also presented, such as those that support preparation for Oracle certifications.

Exercise 8.a) Polymorphism for Methods, True or False:

1. Method overloading implies writing another method with the same name and a different return type.
2. Method overloading implies writing another method with a different name and the same list of parameters.
3. The signature of a method consists of the identifier - parameter list pair.

4. To take advantage of method overriding, inheritance must be in place.
5. To take advantage of method overloading, inheritance must be in place.
6. Suppose that the class `B`, which extends the class `A`, inherits the following method:

```
public int m(int a, String b) { ... }
```

   If in the B class, we write the following method:

```
public int m(int c, String b) { ... }
```

   we are doing method overloading and not overriding.

7. If in the `B` class, we write the following method:

```
public int m(String a, String b) { ... }
```

   we are doing method overloading and not overriding.

8. If in the `B` class, we write the following method:

```
public void m(int a, String b) { ... }
```

   we will get a compilation error.

9. If in the `B` class, we write the following method:

```
protected int m(int a, String b) { ... }
```

   we will get a compilation error.

10. If in the `B` class, we write the following method:

```
public int m(String a, int c) { ... }
```

   we are doing a method overriding.

Exercise 8.b) Polymorphism for Data, True or False:

1. Considering the classes introduced in this chapter, the following code fragment will not produce compilation errors:

```
Vehicle v [] = {new Car(), new Plane(), new Plane()};
```

2. Considering the classes introduced in this chapter, the following code fragment will not produce compilation errors:

```
Object o [] = {new Car(), new Plane(), "ciao"};
```

3. Considering the classes introduced in this chapter, the following code fragment will not produce compilation errors:

```
Plane a [] = {new Vehicle(), new Plane(), new Car()};
```

4. Considering the classes introduced in this chapter, and if the method of the `Traveler` class was the following:

```
public void travel(Object o) {
   o.accelerate();
}
```

we could pass as parameter to it an object of type Vehicle without having compilation errors. For example:

```
claudio.travel(new Vehicle());
```

5. Considering the classes introduced in this chapter, the following code fragment will not produce compilation errors:

```
ThreeDimensionalPoint ogg = new Point();
```

6. Considering the classes introduced in this chapter, the following code fragment will not produce compilation errors:

```
ThreeDimensionalPoint ogg = (ThreeDimensionalPoint)new Point();
```

7. Considering the classes introduced in this chapter, the following code fragment will not produce compilation errors:

```
Point ogg = new ThreeDimensionalPoint();
```

8. Considering the classes introduced in this chapter, and if the `Piper` class extends the `Plane` class, the following snippet will not produce compilation errors:

```
Vehicle a = new Piper();
```

9. Considering the classes introduced in this chapter, the following code fragment will not produce compilation errors:

```
String string = fiat500.toString();
```

10. Considering the classes introduced in this chapter, the following code fragment will not produce compilation errors:

```
public void payEmployee(Employee emp) {
    if (emp instanceof Employee) {
        emp.setSalary(1000);
    } else if (emp instanceof Programmer) {
        ...
    }
```

Exercise 8.c)

To start creating a simple IDE, create a `FileType` interface that defines static constants that represent the types of source files that our IDE should manage. Certainly, one of these constants must be called `JAVA`. Choose all the others as you like. Also choose the type of constants as you like.

Exercise 8.d)

Create a `File` class that abstracts the concept of a generic file and defines a name and a type.

Exercise 8.e)

Create a `SourceFile` class that abstracts the concept of source file (extending the `File` class) which also defines a string type content.

Exercise 8.f)

Add an `addText()` method, that adds a text string to the end of the contents of the source file.

Exercise 8.g)

Add an overloaded `addText()` method, that adds a `text` string to a specified point in the contents of the source file (see the `String` class documentation).

Exercise 8.h)

Create a `SourceFileTest` class that tests the correct operation of the `SourceFile` class.

Exercise 8.i)

Create an `Editor` interface that abstracts the concept of text editor. You need to define methods to open, close, save and edit a file.

Exercise 8.j)

Create an `IDE` interface that abstracts the concept of IDE. Please note that an IDE is also an editor. You need to define methods to compile and execute a file.

Exercise 8.k)

Create a simple `JavaIDE` implementation of the `IDE` interface. Add an implementation for the `edit()` method (and as you wish, you can re-implement all the methods that you find useful).

Exercise 8.l)

Starting from the solutions of the Exercise 7.k, in which we created classes to simulate a bookshop, perform the following steps:

1. Create the `bookshop.data`, `bookshop.business`, `bookshop.util` and `bookshop.interface` packages, and move the existing classes within the packages that are deemed most appropriate.
2. For the `Book` and `Album` classes, create the `toString()` method, which returns a one line description of the item. The `toString()` method should print this information: `article_type: (isbn) - "title" by author, genre, price`. For example: `Book: (2345) "The Pillars of the Earth" by Ken Follet, Romance, 18€`,
3. Modify the `ItemTest` class, so that it also test the `toString()` methods added.

Exercise 8.m)

Starting from the solution of the previous exercise, create a `ShoppingCart` class, which represents a shopping cart. This abstraction must declare the methods: `add()` to add an item to the shopping cart, `calculatePrice()` which returns the total price of the items in the shopping cart, `toString()` which prints the description of the items in the shopping cart. The shopping cart can contain a maximum of four items (books or albums). Make sure that the `add()` method prints an error when trying to add the eleventh article, and that the element is consequently not added.

Create the `ShoppingCartTest` class, to test the correct functioning of the `ShoppingCart` class, invoking all the defined methods, and try adding five items to verify the correct functioning of the check implemented in the `add()` method.

Exercise 8.n)

Create a `IDETest` test class that performs file operations using `IDE`.

The exercise could continue extending these classes further, feel free to do other programming iterations after completing this one. You will have to perform three steps: provide yourself specifications, understand how to implement them and implement them.

Exercise 8.o) Varargs, True or False:

1. The varargs allow you to use the methods as if they were overloads.
2. The following declaration can be compiled without errors:

```
public void myMethod(String... s, Date d) {
  ...
}
```

3. The following declaration can be compiled without errors:

```
public void myMethod(String... s, Date d...) {
  ...
}
```

4. Considering the following method:

```
public void myMethod(Object... o) {
  ...
}
```

   the following invocation is correct:

```
oggetto.myMethod();
```

5. The following declaration can be compiled without errors:

```
public void myMethod(Object o, Object os...) {
  ...
}
```

6. Considering the following method:

```
public void myMethod(int i, int... is) {
  ...
}
```

   the following invocation is correct:

```
object.myMethod(new Integer(1));
```

7. The rules of overriding change with the introduction of varargs.
8. The `printf()` method of the `java.io.PrintStream` class is based on the `format()` method of the `java.util.Formatter` class.
9. The `format()` method of the `java.util.Formatter` class has no overload because it is defined with a varargs.

10. In case you pass an array as varargs to the `printf()` method of `java.io.PrintStream`, this will be treated not as a single object but as if each of its elements had been passed to one by one.

Exercise 8.p)

Given the following hierarchy:

```
interface A {
    void method();
}

interface B implements A {
    static void staticMethod() {}
}

final class C implements B {}

public abstract class D implements A {
    @Override
    void method() {}
}
```

Choose all the true statements:

1. Interface `C` does not inherit the `staticMethod()` method.
2. Interface `B` cannot implement another interface.
3. The class `C` implementing `B` also inherits the abstract `method()` method of the `A` interface, and since it is not declared `abstract` it cannot be compiled.
4. Class `D` cannot be compiled because it cannot be declared `abstract`. In fact, it does not declare any abstract method.
5. Class `D` does not compile because the method it declares is not declared `public`.
6. Class `D` compiles only because the method is annotated with `Override`.

Exercise 8.q)

Which of the following statements is correct (choose all that apply):

1. An interface extends the class `Object`.
2. A method that takes as a parameter a reference of type `Object`, can take as input any object of any type, even of an interface type.

3. A method that takes as a parameter a reference of type `Object`, can take as input any object of any type, even an array.
4. A method that takes as a parameter a reference of type `Object`, can take as an input any object of any type, even a heterogeneous collection.
5. All casts of objects are evaluated at compile time

Exercise 8.r)

Consider the following classes:

```
public class PrintNumber {
    public void print(double number){
        System.out.print(number);
    }
}

public class PrintInteger extends PrintNumber{
    public void print(int number) {
        System.out.print(number);
    }

    public static void main(String args[]) {
        PrintNumber printNumber = new PrintInteger();
        printNumber.print(1);
    }
}
```

If we run the `PrintInteger` class, what will be the output?

1. 1.2
2. 1
3. 1.0
4. 11.2

Exercise 8.s)

Consider the following hierarchy:

```
public interfac7e Satellite {
    void orbit();
}
```

```
public class Moon implements Satellite{
    @Override
    public void orbit(){
        System.out.println("Moon is orbiting");
    }
}

public class ArtificialSatellite implements Satellite {
    @Override
    public void orbit() {
        System.out.println("Artificiale satellite is orbiting");
    }
}
```

And the following class of tests:

```
public class SatellitesTest {
    public static void main(String args[]) {
        test(new Moon(), new ArtificialSatellite());
        Satellite[] satellites = {
            new Moon(), new ArtificialSatellite()
        } ;
        test(satellites);
        test();
      //  test(new Object());
    }

    public static void test(Satellite... satellites) {
        for (Satellite satellite : satellites) {
            satellite.orbit();
        }
    }
}
```

Choose all the correct statements:

1. The application compiles and runs without errors.
2. The application does not compile because of the instruction `test(satellites);`.
3. The application does not compile for the instruction `test();`.
4. The application does not compile for the instruction `test(new Object());`.
5. The application compiles but it, due to an exception, crashes at runtime.

Exercise 8.t)

Define the overload and the override concepts. And give an example of a subclass, which implements both concepts.

Exercise 8.u)

Bearing in mind that `Number` is superclass of the `Integer` class, let us consider the following hierarchy:

```
public abstract class SumNumber {
    public abstract Number sum(Number n1, Number n2);
}

public class SumInteger extends SumNumber{
    @Override
    public Integer sum(Number n1, Number n2) {
        return (Integer)n1 + (Integer)n2;
    }
}
```

Choose all the correct statements:

1. The `SumInteger` class compiles without errors.
2. The `SumInteger` class does not compile because the override is not correct: the return types do not coincide.
3. The `SumInteger` class does not compile because it is not possible to use the + operator except with primitive type numbers.
4. The `SumInteger` class could cause an exception at runtime.

Exercise 8.v)

Make the `sum()` method of the `SumInteger` class defined in exercise 8.u robust, so that the runtime works without exceptions.

Exercise 8.w)

Briefly define what a polymorphic parameter is, what heterogeneous collections are, and what a virtual call to a method is.

Exercise 8.x)

Starting from the solution of the Exercise 7.y, let's evolve our application that simulates a phone book:

1. Replace the `printDetails()` method with the override of the `toString()` method, in the `Contact` and `Special` classes. This change is consistent, because these classes represent application data, and therefore should not "print details". Make sure that the `toString()` method also returns the identifier (`id` variable).

2. In the PhoneBook class, add the elements of the specialContacts array to the contacts array, and delete the specialContacts variable. Modify the getContacts() and getSpecialContacts() methods to make that the first return all contacts (both special and ordinary contacts), while the latter returns only the special contacts.
3. If necessary, modify the User class to make it work consistently with the new implementation of the PhoneBook class.
4. Modify the SearchContacts, and SearchSpecialContacts classes, so that they work in line with the changes made.

Exercise 8.y)

1. Starting from the solution of the previous exercise:
2. Add in the PhoneBook class a method called getOrdinaryContacts() that returns all ordinary contacts, that is, the contacts that are not of the Special type.
3. Add a method called searchOrdinaryContactsByName() in the User class, which searches by name only contacts that are not special.
4. Create a class SearchOrdinaryContacts, homologous to the other search classes we have already created.

Exercise 8.z)

1. Starting from the solution of the previous exercise:
2. Until now, we have assigned the responsibility for creating Contact objects explicitly to the class containing the main() method. So, now is the time to create a ContactFactory class that has this responsibility, declaring one or more static methods getContact(). These methods will take as input the necessary fields to instantiate the objects.
3. In the User class, add a method called add() which adds a Contact object to the contacts array. Pay attention that in the PhoneBook class, the contacts array can contain a finite number of Contact objects. Add a number of free places in the array to be filled with new objects.
4. Test with the SearchContacts class the correct functioning of the add() method of the PhoneBook class. Create the objects to pass to the method using the ContactFactory class.