Practical Work of Languages, Technologies, and Paradigms of Programming

2018-19

Part I
Programming in Java

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Practice 3 - Prior reading material

Genericity in Java

Contents

1 Generic classes **2**

2 Genericity and inheritance **3**

1 Generic classes

A generic class is a class where its declaration is parametrized by *type variables* also known as *generic types* or *generic variables* to distinguish them from *pure types* (non-generic or concrete types).

The name of generic variables are written after the name of the class and are separated by commas. They are written between `<` and `>`. These variables represent types which are not *basic or primitive types* but, rather, classes (*reference types*).

`class/interface ClassName` <list-of-generic-types> `{ ... }`

By convention, these variables are written in uppercase.

Let us see a brief introduction to the use of generic classes by defining and using the class `G1`, we will see later the use of a predefined generic class.

```
public class G1<T> {
  private T a;
  public G1(T x) { a = x; }
  public String toString() { return `` '' + a; }
}
```

where `T` is the name of a variable which does not represent a value but a type. This variable is used inside the class in the same way as a pure (or concrete) type, as can be observed in line 2 where the attribute `a` has type `T`. Observe that `<...>`, sometimes found between the class name and parameters, is not used in the definition of the constructor. However, the construction of objects uses the following syntax:

`new ClassName` <list-of-pure-types> `(`list-of-formal-parameters`)`

The replacement of generic variables by a type is done during the static analysis of the code. Since objects are created dynamically, only concrete types can be used to create them. Otherwise stated, there are generic classes but not generic objects. The list-of-types cannot contain basic or primitive types (such as `byte`, `short`, `int`,...) which are not objects in Java. We can use instead their corresponding *wrapper classes* studied below and that will be used in the first exercise of this practice.

For instance, we can create an object of the class `G1` containing one `String` in the attribute `a` by using the constructor with:

```
new G1<String>("hello world")
```

When this object is created, the compiler replaces the generic variable `T` by the class `String`. This type is assigned to the attribute `a` in line 2. Similarly, it is assigned to the parameter of the constructor in line 3. It is roughly as if we have written the class `G1` as follows:

```
private String a;
public G1(String x) { a = x; } ...
```

Besides, Java provides generic classes already defined. Among them, the `ArrayList` class that implements a resizable array whose elements are of the generic type `<T>` that will be replaced by a concrete type when making an instance. For example:

```
new ArrayList<String>();
```

The `ArrayList` class will be used in the second exercise of this practice.

2 Genericity and inheritance

The Java virtual machine does not handle objects of generic type. Not surprisingly, the information about type variables used to parametrize the generic type is lost at execution time. Each type variable is transformed into a variable of a concrete type during the static analysis at compile time. This has some consequences. For instance, the use of `instanceof` is restricted to concrete/non-generic/pure types. The following example shows that the type variable `T` can only be used if it is a parameter of the class where it is used:

```
class ClaseX<T> {
  ...
  if (variable instanceof ClaseA<T>)
  ...
}
```

Besides the possibility of defining generic classes, as seen in previous examples, it is also possible to define generic instance methods and generic static methods in any class, even if it is not a generic one. To this end, the type returned by the method is preceded by the type variables to be used in the definition of the method. The following example defines two classes and, in one of them (in the `Test` class) there is a call to the generic method `metodo`.

```
import java.util.*;
class Estaticos {
  public static <T> void metodo(ArrayList<T> p) {
    System.out.println(p);
  }
}

public class Test {
  public static void main(String[] args) {
    ArrayList<Figure> lf = new ArrayList<Figure>();
    lf.add(new Circle(1, 2, 3));
    Estaticos.metodo(lf);
  }
}
```

As you know, given two (pure or non-generic) classes (or interfaces) `ClassB` and `ClassA`, where the first one extends the second one, we say that the first one is *compatible* with the second one. This means that it is possible to assign an object of type `ClassB` to a variable of type `ClassA`.

Using the example from practice 1, it is possible to assign an object of type `Figure` or any other type derived from it (e.g. `Triangle`) to a variable of type `Figure`. It is also possible to assign any value of a type implementing the `Printable` interface to a variable of this type (`Printable p = new Rectangle(1, 2, 3, 4);`).

However, this compatibility of types can be lost with genericity. Let us suppose that the following generic class `ClaseX<T>` is particularized with the following pure classes `ClaseX<ClaseA>` and `ClaseX<ClaseB>`. The second one is not a subclass of the first one (despite of the fact that `ClaseB` extends `ClaseA`). In this way, if we try to compile the following code:

```
1 class ClaseB extends ClaseA {}
2 class ClaseA {}
3 class ClaseX<T> {}

4 class TestGenericidadYHerencia {
5   public static void main(String[] args) {
6     ClaseX<ClaseA> cXA1 = new ClaseX<ClaseA>();
7     ClaseX<ClaseA> cXA2 = new ClaseX<ClaseB>();
8   }
9 }
```

we can observe that an error is produced in line 7 due to a lack of compatibility in the assignment, even if `ClaseB` inherits from `ClaseA`.

The joint use of genericity and inheritance allows us to consider several possibilities. Observe that the following examples are based on the extension of classes, but this could also be applied to implementation and extension of interfaces:

- Do not propagate genericity, just to define a concrete class (`StringList`) from a generic type with a pure or concrete type (`ArrayList<String>`).

  ```
  class StringList extends ArrayList<String> {...}
  ```

- To maintain genericity in the parent class (`ArrayList`) by maintaining the list of type variables. For instance:

  ```
  class SortedList<T> extends ArrayList<T> {...}
  ```

- To restrict genericity of one of the type variables by writing, after this variable, the reserved word `extends` followed by the type which we want to restrict. As can be observed in the following example where a list is restricted to objects which must extend `Figure`:

  ```
  class FiguresList<T extends Figure> extends ArrayList<T> {...}
  ```

  which can be used to create a list of circles as follows:

  ```
  FiguresList<Circle> l = new FiguresList<Circle>();
  ```

  It is also possible to restrict with generic types, as can be observed in the following examples which define a list of generic stacks:

```
class StackList1<T extends Stack<K>, K> extends ArrayList<T> {...}
class StackList2<K> extends ArrayList< Stack<K> > {...}
```

- To increase the genericity by including other type variables than those inherited from the parent, as illustrated in the following example:

```
class PlusList<T, G> extends ArrayList<T> {...}
```