Practical Work of Languages, Technologies, and Paradigms of Programming

2018-19

Part I
Programming in Java

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Practice 3

Genericity in Java

Contents

1 Wrapper classes

As you know, Java programming language has primitive data types (`int`, `double`, `boolean`, etc.) which are not objects. It is sometimes convenient to treat these primitive data as if they were objects. This makes it possible to use them in a generic way. For example, containers defined by the API in the package `java.util` (dynamic arrays, linked lists, collections, sets, etc.) are all defined in a generic way using type variables.

Hence, these containers can be instantiated to store objects of a determined type for any desired object type. However, primitive data are not objects, so they are excluded in principle.

In order to solve this situation, the Java API includes what is known as *wrapper classes*. Their function is, in essence, what the name "wrapper" suggest: to allow the use of primitive types as if they were objects. We could define, for instance, such a wrapper class for integers in a quite simple way:

```
public class Integer {
  private int valor;
  public Integer(int valor) { this.valor = valor; }
  public int intValue() { return this.valor; }
}
```

The Java API already provides the set of wrapper classes associated to each primitive type, making it unnecessary the task of writing these classes ourselves. In addition to the basic functionality illustrated in the previous example, wrapper classes contain useful methods to manipulate primitive types (casting/coercion from and to these primitive types, conversion to String, etc.).

Each of Java's eight primitive data types has a wrapper class dedicated to it. The wrapper classes are: `Byte` for `byte`; `Short` for `short`; `Integer` for `int`: `Long` for `long`; `Boolean` for `boolean`; `Float` for `float`; `Double` for `double` and `Character` for `char`. Wrapper classes are part of the `java.lang package`, which is imported by default into all Java programs.

When we create an instance of a generic class, we cannot instantiate the type variable with a basic type, we should use a wrapper class instead. For example, if we want to instantiate the generic class `G1<T>` with integers, we can create a new instance by calling `new G1<Integer>(...)`. When this object is created, the compiler replaces the generic variable `T` by `Integer`.

Exercise 1 *Create a program in the* `main` *method of the* `WrapperClassesUse` *class (partially implemented, available in Poliformat) where a variable is defined for each basic type in Java. Then, assign to each variable a reference to an instance of its corresponding wrapper class. Write the contents of the variables in the standard output. In the same* `main`*, do the same in the inverse direction: define variables of wrapper types and assign its corresponding value to the basic type. Print the value of all the variables on the standard output.*

2 Predefined generic classes in Java

There are many generic classes predefined in Java. One of them is the class `ArrayList`, which implements a resizable array. The hierarchy of this generic class is illustrated in Figure 1, obtained from Java API, where we can observe parent classes and other useful information.

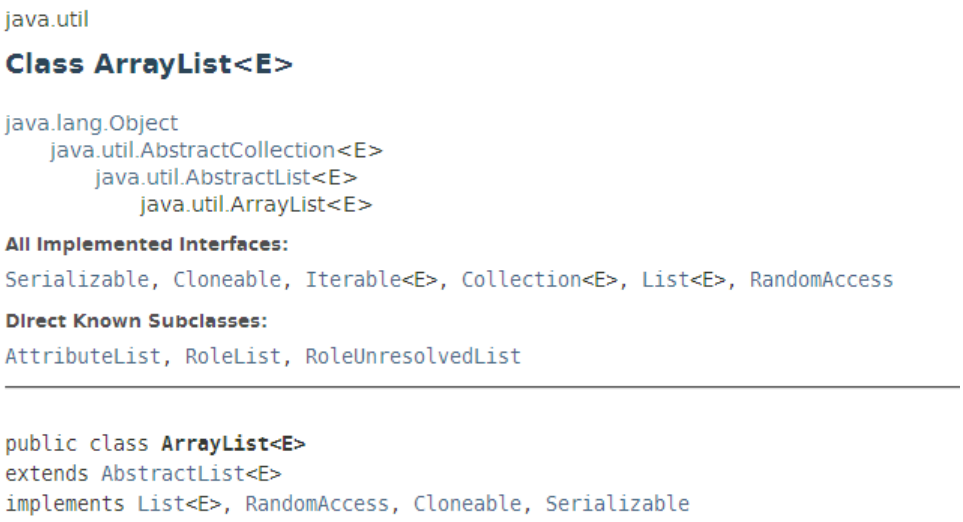

Figure 1: `ArrayList<E>` Hierarchy

This class is contained in the package `java.util`. It extends the abstract and generic class `AbstractList<E>`, which is a subclass of the (also abstract and generic) class `AbstractCollection<E>`. This last class descends, in turn, from `Object` which is stored in the package `java.lang` (this package contains the kernel of the language and is always imported by default).

The `ArrayList` class implements also six interfaces (three of them generic) including `List<E>`, which is also implemented by `AbstractList<E>` as can be observed by checking its API.

The `ArrayList<E>` class does not implement all methods specified in the `List<E>` interface because the parent class of `ArrayList<E>` already implements methods of the same interface.

Many of these classes are stored in the package `java.util`, but not all. The three classes extending `ArrayList<E>` which are predefined in the language can be found in different packages, as is the case of the three interfaces that they implement.

Exercise 2 *Complete the code of the* `ArrayListUse` *class (partially implemented, available in Poliformat) to read lines from a file and display them sorted alphabetically. Perform the following steps in its* `main` *method:*

- *Create an object of the class* `File` *with the name of a text file you create as a parameter and link it with the variable of the same type* `fd`*. For reading the file, define a variable* `fichero` *of type* `Scanner` *and create an instance of this class passing the variable* `fd` *as an argument to the constructor. Create*

an instance of the class `ArrayList<E>` *with the type* `String` *and link it with the variable* `list` *of the same type.*

- *File reading can be done by means of a* `while` *loop that checks in its guard whether the file still contains something. This check can be done by using the method* `hasNext()` *of the variable* `fichero`*. In the body of the loop, read a text line by using the method* `nextLine()` *on the same variable. This text line can be added to the object of type* `ArrayList<String>` *by passing it as an argument to the method* `add(E e)` *applied to* `list`*.*
- *Sort the lines of the list with the static method* `sort(List<T> list)` *of the class* `java.util.Collections`*. This method may receive as a parameter those objects whose class implements the interface* `List<E>`*. Among these classes, we can find the class* `ArrayList<E>`*.*
- *Write the strings stored in* `list` *by using the method* `toString` *which is defined in the class* `ArrayList`*.*

You can find more information on the use of these methods in the API.

3 Implementation of the generic type `Queue<T>`

Linear types are those whose elements are formed by linearities or sequences which have query and modification operations.

A queue is a FIFO (*First In, First Out*) linear structure where the first inserted element is the first leaving it. Figure 2 shows the specification of the `Queue<T>` type.

You can find in Poliformat the main directory where the application, called `lineales`, is stored. This directory has a package called `librerias` which contains three subdirectories/packages:

- `librerias.modelos` contains an interface specifying the operations associated to queues in the interface `Queue<T>`.
- `librerias.implementaciones` contains two partial implementations of the interface.
- `librerias.aplicaciones` contains programs which make use of the `Queue<T>` data type.

Exercise 3 *Uncompress the file* `lineales.rar` *(available in Poliformat). You will find in this project the file* `QueueAC.java` *partially implemented. There are methods already implemented and others that you have to complete.*

You also have to complete the declaration of the attributes. You have to implement the queues using circular arrays in the `QueueAC<T>` *class, as shown in Figure 3. The internal structure of an object of type* `QueueAC <T>` *must have at least:*

- *an attribute,* `theArray`*, which is an array of the generic type* `T` *to save the elements of the queue.*
- *two attributes,* `first` *and* `last`*, of type* `int` *to refer the indices where the first and last elements of the queue are.*
- *an attribute,* `size`*, to represent the number of elements of the queue.*

librerias.modelos

Interface Queue<T>

```
public interface Queue<T>
```

interface Queue it defines the TAD of a generic queue

Method Summary	
abstract T	**dequeue**() Queries and extracts the first element, only if the queue is not empty
abstract void	**enqueue**(T e) Inserts the element at the end of the queue
abstract T	**first**() Queries the first element, in order of insertion, only if the queue is not empty
abstract boolean	**isEmpty**() Verifies if the queue is empty
abstract int	**size**() Queries the number of elements of the queue

Figure 2: `Queue<T>` interface

The private method `int increase(int i)` *is responsible for returning the position next to* `i` *considering the array as if it were circular.*

Once you have completed the class, check your code by executing the `QueueApp` *class from the* `aplicaciones` *library.*

Exercise 4 *Now, you have to implement a resizable queue. You will find, in the same project, the file* `QueueAL.java` *partially implemented. You have to complete the implementation taking into account that the internal data structure must be an instance of the* `ArrayList<T>` *class.*

In order to implement the methods, you have to use the operations specified in the API of the `ArrayList` *class (among them, you can find how to add and remove elements from the list, get the size, etc.).*

Once completed the class, check your code by modifying first (in order to use the new implementation) and executing later the `QueueApp` *class from the* `aplicaciones` *library.*

Exercise 5 *Consider that you want to implement a resizable queue whose elements can only be instances of the* `Figure` *class or any of its subclasses. A basic implementation would be the following one:*

```
class FiguresQueue<T extends Figure> extends QueueAL<T> { }
```

Taking into account this implementation, identify in the following program which lines would lead to compilation errors and explain why.

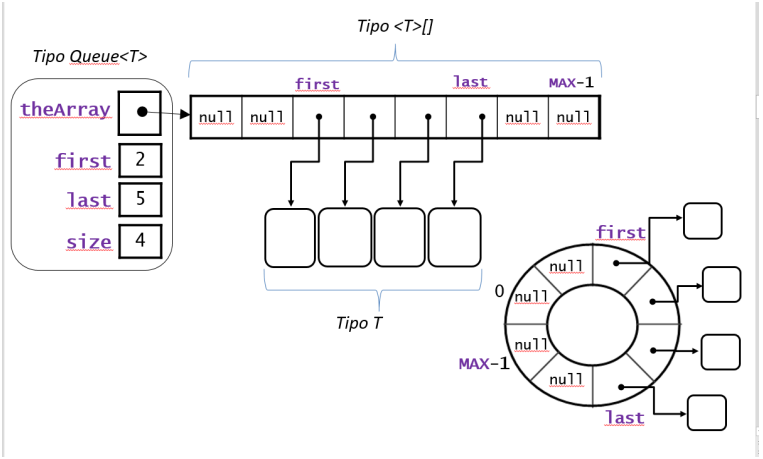

Figure 3: Data structure of the `QueueAC<T>` class

```
public static void main(String[] args) {
  Queue<String> a = new FiguresQueue<String>();
  Queue<Object> b = new FiguresQueue<Object>();
  Queue<Circle> c = new FiguresQueue<Circle>();
  Queue<Figure> f = new FiguresQueue<Figure>();
  for (int i = 1; i <= 9; i++) {
    c.enqueue( new Circle(0, 0, i) );
    c.enqueue( new Triangle(0, 0, i, i) );
    c.enqueue( new Integer(i) );
  }
  for (int i = 1; i <= 9; i++) {
    f.enqueue( new Circle(0, 0, i) );
    f.enqueue( new Triangle(0, 0, i, i) );
    f.enqueue( new Integer(i) );
  }
}
```