Prácticas de Lenguajes, Tecnologías y Paradigmas de programación. Curso 2019-20

Parte I: Java

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Práctica 3

Genericidad en Java

Índice

1. Clases envoltorio

Con frecuencia será útil poder tratar los datos primitivos (`int`, `double`, `boolean`, etc.) como objetos pudiendo así, además, utilizarlos genéricamente. Por ejemplo, todos los contenedores definidos por el API de Java en el package `java.util` (arrays dinámicos, listas enlazadas, colecciones, conjuntos, etc.) están definidos en términos genéricos con variables de tipo.

Estos contenedores pueden almacenar, por lo tanto, cualquier tipo de objetos. Pero los datos primitivos no son objetos, y, en principio, quedan excluidos de estas posibilidades.

Para resolver esta situación el API incorpora las clases envoltorio (*wrapper classes*), que consisten en dotar a los datos primitivos con un envoltorio que permita tratarlos como objetos. Por ejemplo, podríamos definir una clase envoltorio para los enteros, de forma bastante sencilla, con:

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

La API hace innecesario esta tarea al proporcionar un conjunto completo de clases envoltorio para todos los tipos primitivos. Adicionalmente a la funcionalidad básica que se muestra en el ejemplo, las clases envoltorio proporcionan métodos de utilidad para la manipulación de datos primitivos (conversiones de y hacia datos primitivos, conversiones a String, etc.).

Las clases envoltorio existentes son: `Byte` para `byte`; `Short` para `short`; `Integer` para `int`; `Long` para `long`; `Boolean` para `boolean`; `Float` para `float`; `Double` para `double` y `Character` para `char`.

Cuando creamos una instancia de una clase genérica, no se pueden usar tipos básicos como `int` como instancia del tipo genérico, y entonces hay que usar las correspondientes *clases envoltorio*. Por ejemplo, podemos crear un objeto de una clase genérica `G1<T>` para que contenga un `int` en su atributo invocando al constructor de la clase con `new G1<Integer>(...)`. Al crear este objeto, la *variable de tipo* `T` ya ha sido cambiada por el compilador al tipo de la clase envoltorio `Integer`.

Ejercicio 1 *Escribe un programa en el método* `main` *de la clase* `WrapperClassesUse` *(implementada parcialmente, disponible en Poliformat) en el que se definan variables para cada tipo básico. Asigna a cada variable un objeto de su correspondiente clase envoltorio. Escribe el contenido de las variables en la salida estándar. En el mismo* `main`*, haz lo mismo en sentido inverso: define variables de tipos envoltorio y asígnales su correspondiente valor de tipo básico.*

2. Clases genéricas predefinidas

Existen multitud de clases genéricas predefinidas en Java. Una de ellas es la clase `ArrayList`, que implementa un array redimensionable. En la Figura 1, extraída de las APIs de Java, se representa la jerarquía de clases de la que desciende.

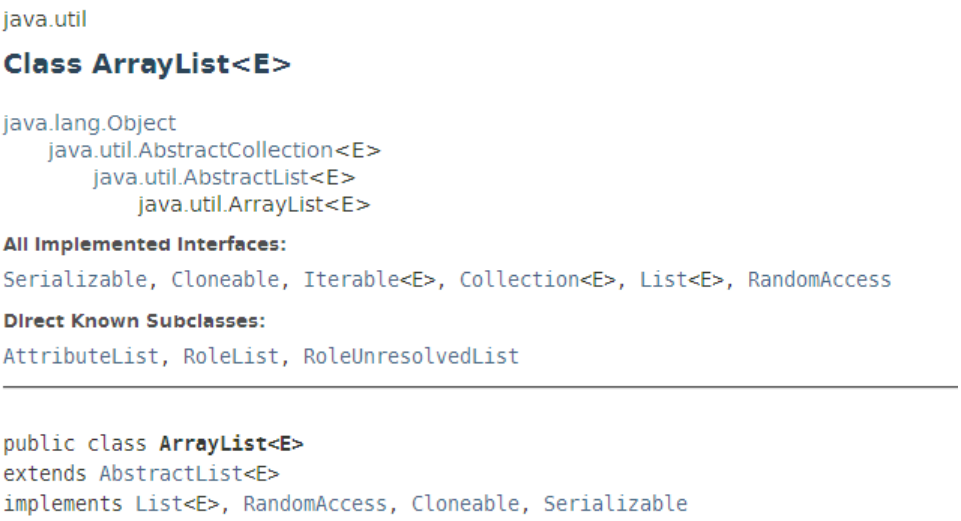

Figura 1: Jerarquía de `ArrayList<E>`

Esta clase está contenida en el paquete `java.util`. Extiende de la clase abstracta y genérica `AbstractList<E>`, la cual deriva de la clase `AbstractCollection<E>` (también abstracta y genérica), que a su vez desciende de `Object` que se encuentra en el paquete `java.lang`, el cual contiene el núcleo del lenguaje y siempre se importa por defecto.

También implementa seis interfaces (tres de ellas genéricas) entre las que se encuentra la interfaz `List<E>`, la cual, siguiendo su API, también es implementada por la clase `AbstractList<E>`.

La cantidad de métodos definidos en la clase `ArrayList<E>` es menor que los que especifica la interfaz `List<E>`. Esto se explica porque la clase padre de `ArrayList<E>` implementa métodos de la misma interfaz.

Muchas de estas clases están en el paquete `java.util`, pero no todas. Las tres clases predefinidas en el lenguaje que extienden de `ArrayList<E>` están en distintos paquetes, al igual que tres de las interfaces que implementa.

Ejercicio 2 *Completa el código en la clase* `ArrayListUse` *(implementada parcialmente, disponible en Poliformat) para que lea líneas de un fichero y las muestre ordenadas alfabéticamente. En su método* `main` *realiza los siguientes pasos:*

- *Crea un objeto de la clase* `File` *con el nombre del fichero como parámetro y referéncialo con la variable del mismo tipo* `fd`*. Para la lectura del fichero, define la variable* `file` *de tipo* `Scanner` *y crea una instancia de esta clase pasando la variable* `fd` *como argumento al constructor. Crea una instancia de la clase* `ArrayList<E>` *con el tipo puro* `String` *y referénciala con la variable* `list` *del mismo tipo.*
- *La lectura se hará con un bucle* `while` *que compruebe en su guarda que no se ha llegado al final del fichero aplicando el método* `hasNext()` *a la variable* `file`*. En el cuerpo del bucle lee una línea del texto aplicando el método* `nextLine()` *a la misma variable. Para añadir la linea al objeto de tipo* `ArrayList<String>`*, pásala como argumento al método* `add(E e)` *aplicado a* `list`*.*

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

Figura 2: Interfaz `Queue<T>`

- *Ordena las líneas de la lista con el método estático* `sort(List<T> list)` *de la clase* `java.util.Collections`*. Este método recibe como parámetro objetos cuya clase implemente el interfaz* `List<E>`*. Entre estas clases se encuentra la clase* `ArrayList<E>`*.*
- *Escribe las cadenas de caracteres guardadas en* `list` *invocando el método* `toString` *que por defecto está definido en la clase* `ArrayList`*.*

Puedes encontrar más información sobre el uso de los métodos en la API.

3. Implementación del tipo genérico `Queue<T>`

Como sabes, los tipos de datos lineales son aquellos cuyos elementos están formados por linealidades o secuencias a las que se les puede aplicar operaciones de modificación y consulta.

Una cola es una estructura lineal FIFO (*First In, First Out*) en la que el primer elemento que entra es el primero que sale. La especificación del tipo `Queue<T>` se describe en la Figura 2.

En Poliformat puedes encontrar el directorio principal donde se guarda la aplicación que se llama `lineales`. Este directorio tiene un subdirectorio (paquete Java) `librerias`, el cual contiene, a su vez, tres subdirectorios correspondientes a los tres paquetes:

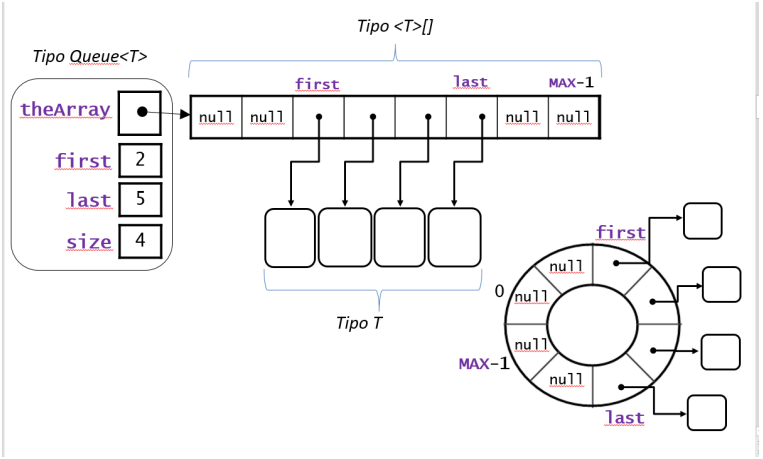

Figura 3: Estructura de datos de la clase `QueueAC<T>`

- `librerias.modelos`, que contiene la especificación de las operaciones de las colas en la interfaz `Queue<T>`.
- `librerias.implementaciones`, que contiene dos implementaciones parciales de la interfaz.
- `librerias.aplicaciones`, que contiene programas que usan el tipo `Queue<T>`.

Ejercicio 3 *Descomprime el fichero* `lineales.rar` *(disponible en Poliformat). En el proyecto, encontrarás el fichero* `QueueAC.java` *parcialmente implementado. Este fichero contiene métodos ya implementados y otros que debes completar.*

También has de completar la declaración de sus atributos. En la clase `QueueAC<T>` *tienes que implementar las colas usando arrays circulares tal y como se ilustra en la Figura 3. Por ello, la estructura interna de un objeto de tipo* `QueueAC<T>` *ha de tener por lo menos:*

- *un atributo,* `theArray`*, array de tipo genérico* `T` *para guardar los elementos de la cola.*
- *dos atributos,* `first` *y* `last`*, de tipo entero para referenciar los índices donde están situados el primer y último elemento de la cola.*
- *un atributo,* `size`*, para representar la cantidad de elementos de la cola.*

El método privado `int increase(int i)` *se encarga de devolver la posición siguiente a* `i` *considerando el array como si fuera circular.*

Una vez completada la clase, comprueba tu código ejecutando la clase `QueueApp` *en la libreria* `aplicaciones`*.*

Ejercicio 4 *Se desea implementar una cola redimensionable. En el mismo proyecto, encontrarás el fichero* `QueueAL.java` *parcialmente implementado. Has de completar su implementación, estableciendo que la estructura de datos interna, soporte de la cola, sea una instancia de la clase* `ArrayList<T>`*.*

Para escribir el codigo de los métodos has de usar las operaciones que se encuentran especificadas en la API de la clase `ArrayList` *(entre las que puedes encontrar cómo añadir y eliminar elementos en la lista, consultar el tamaño de la lista, etc).*

Una vez completada la clase, comprueba tu código modificando primero (para poder utilizar la nueva implementación) y ejecutando después, la clase `QueueApp` *de la libreria* `aplicaciones`.

Ejercicio 5 *Considera que se quiera implementar una cola redimensionable cuyos elementos solamente puedan ser instancias de la clase* `Figure` *o de cualquier subclase de* `Figure`*. Una implementación básica sería la siguiente:*

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

Teniendo en cuenta esta implementación, identifica en el siguiente programa las líneas que darían error de compilación y razona por qué.

```
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) );
  }
}
```