Prácticas de laboratorio de LTP
(Parte I : Java)

Práctica 2: Genericidad

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Jose Luis Pérez
jlperez@dsic.upv.es

Introducción : Clases genéricas

Una **clase genérica** es una clase normal, salvo que su declaración se parametriza con variables de tipo, también llamadas **tipos genéricos** o **variables genéricas,** en contrapartida a los **tipos puros**.

```
public class G1<T>
    private T a;
    public G1(T x) { a = x; }
    public String toString() {
            return  " " + a;  }
```

G1 es una clase genérica con la variable de tipo **T**

a es una variable de tipo **T**

x es una variable de tipo **T**

T es una variable que **no representa un valor** sino un **tipo**, y se escribe dentro de la clase como si se tratara de un tipo puro.

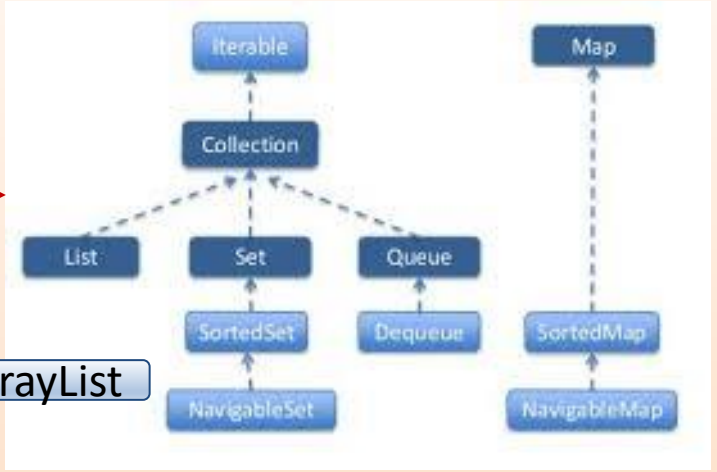
Por ejemplo, podemos crear un objeto de la clase **G1** invocando al constructor de la clase con:

```
G1<String> V1 = new G1<String>("hola mundo");
```

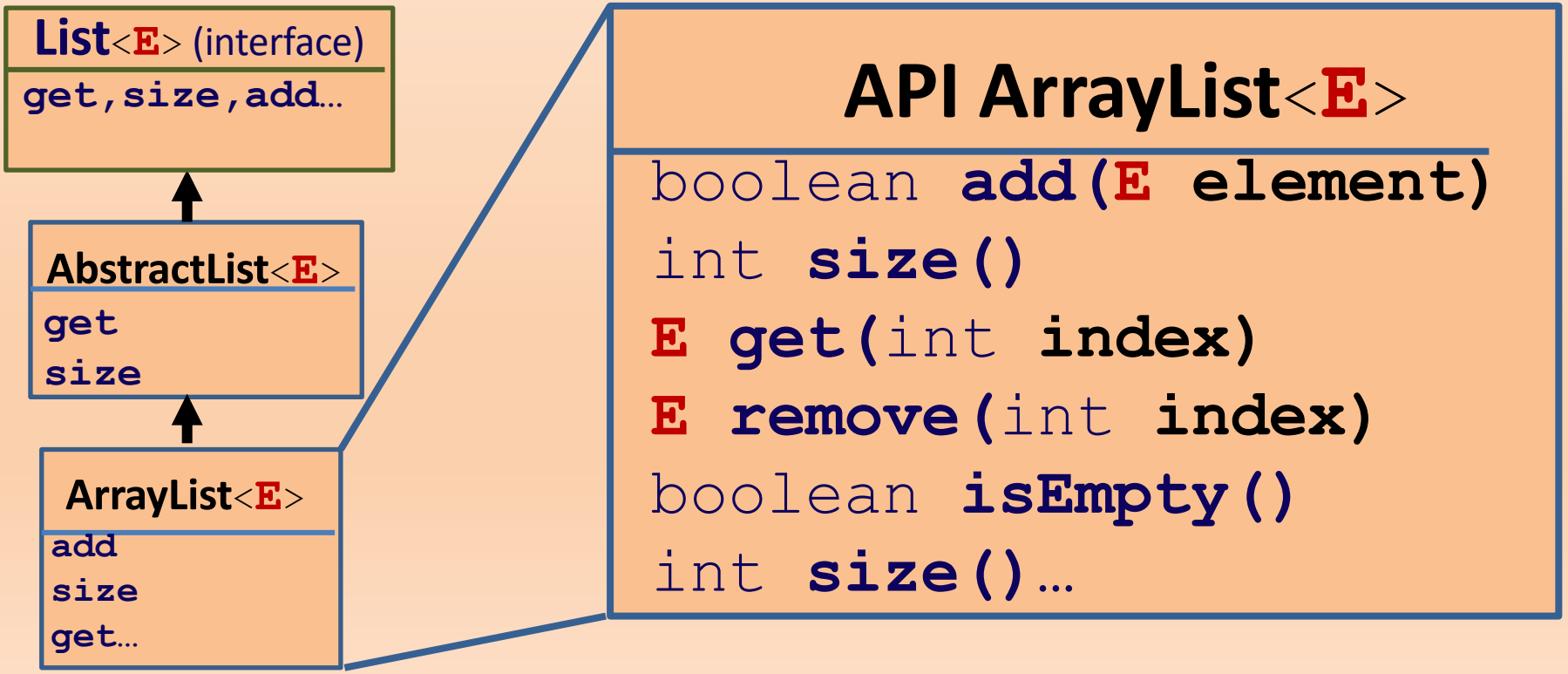
En realidad el atributo de **V1** será de tipo **string**

Introducción : Genericidad y herencia

Java proprorciona **clases genéricas ya predefinidas**. El API de Java en el paquete **java.util** está definido en términos genéricos con variables de tipo. Dentro de este paquete se encuentra la clase **`ArrayList`**, que implementa un array redimensionable:




```
public class ArrayList<E> extends AbstractList<E> implements List<E>…
```



Introducción : Genericidad y herencia

```
public class ArrayList<E> extends AbstractList<E> …
```

En este ejemplo se define la variable **cars** instanciando el tipo genérico como **String**.

Los valores del tipo genérico **E** de **ArrayList** se sustituirán por **String** en tiempo de compilación al instanciar la nueva variable.

```
import java.util.ArrayList;

public class MyClass {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    System.out.println(cars);
  }
}
```

Para poder utilizar **ArrayList** se debe importar de **Java.util**

La variable **cars** es un **ArrayList** de **String**

El método **add** de **ArrayList** permite añadir un nuevo coche al final de la variable **cars**

Introducción : Métodos genéricos

Es posible también definir **métodos de instancia genéricos** y **métodos estáticos genéricos** en cualquier clase, aunque no sea genérica.
En este ejemplo los valores del tipo genérico de **`ArrayList`** se sustituirán por **`Figure`**:

Para poder utilizar la clase **`Figure`** se debe importar el paquete **`practica1`**

```
import java.util.*;
import …/practica1.*;

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); }
}
```

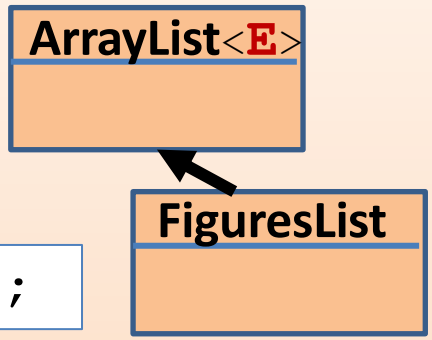
`metodo` Es un método estático genérico

Se instancia la variable genérica de **`ArrayList`** como una **`Figure`**

Se instancia la variable genérica **`T`** de **`Estaticos.metodo`** como una **`Figure`**

Introducción : Genericidad y herencia

Haciendo un uso conjugado de genericidad y herencia podemos plantearnos varios casos para crear nuevas clases:



a) No se propaga la genericidad y se define una clase pura (**FiguresList**) a partir de una genérica con tipo puro **Figure** :

```
class FiguresList extends ArrayList<Figure> { … };
```

En este caso se podría definir una lista de figuras y añadir un triangulo:

```
FiguresList lf = new FiguresList()
lf.add(new Triangle (2,2,3,5));
```

b) Se puede mantener la genericidad pero restringiéndola a una lista de objetos cuyo tipo necesariamente debe extender. En este caso **Figure** :

```
class FiguresList<T extends Figure> extends ArrayList<T> { … };
```

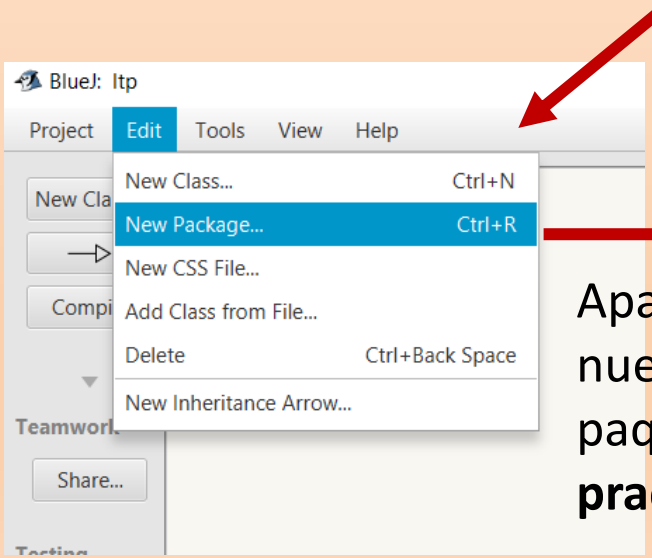
En este caso se podría definir, específicamente, una lista de círculos, añadir un círculo, pero no añadir un triángulo o un objeto **Figure**:

```
FiguresList<Circle> lc = new FiguresList<Circle>()
lc.add(new Circle (2,3,5)); COMPILA CORRECTAMENTE
lc.add(new Figure (2,3));   ERROR COMPILACIÓN
lc.add(new Triangle (2,2,3,5)); ERROR COMPILACIÓN
```

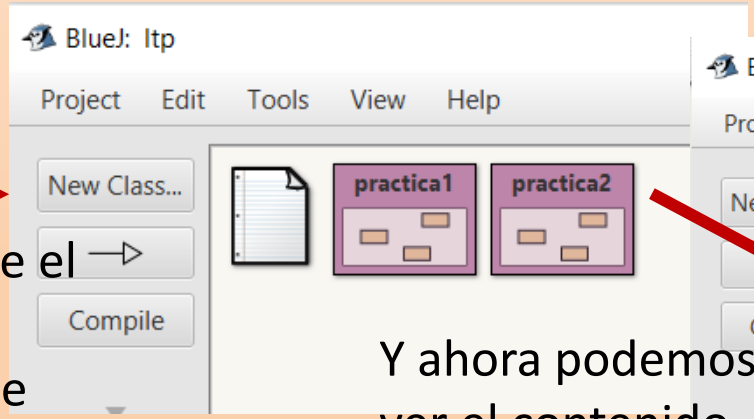
Añadir un nuevo paquete en el proyecto **ltp**

Ejercicio 1: En el proyecto BlueJ **ltp** (de la sesión anterior), crea un paquete de nombre **practica2**. Añade a este paquete las clases **`WrapperClassesUse`** y **`ArrayListUse`** (implementadas parcialmente, disponibles en Poliformat).

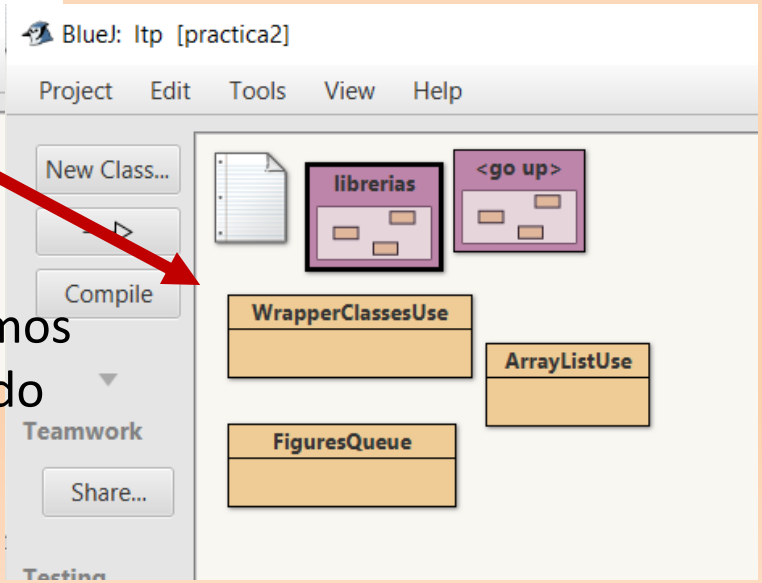
La forma más sencilla de realizar este ejercicio es copiando los ficheros proporcionados para esta práctica en el directorio **practica2,** dentro de la carpeta **ltp** creada en la sesión anterior.
Abrir dicho proyecto y utilizando la opción **"New Package"** seleccionar el directorio **practica2:**



Aparece el nuevo paquete **practica2**



Y ahora podemos ver el contenido del paquete **practica2**



1. Clases envoltorio

Las clases en **`java.util`** están definidas en términos genéricos mediante variables de tipo, que no pueden referenciar a los **tipos primitivos** (`int`, `char`, `double`, etc.), ya que **NO** son objetos.

Por ejemplo, **NO** podemos crear un objeto de la clase genérica **`G1<T>`**, definida anteriormente, invocando al constructor de la clase con `int`:

```
G1<int> V1 = new G1<int>(5);
```

ERROR COMPILACIÓN

Para resolver esta situacion 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; }
}
```

```
G1<Entero> V1 = new G1<Entero>(5);
```

COMPILA CORRECTAMENTE

WrapperClassesUse

Clases envoltorio predefinidas en Java

Ejercicio 2: Escribe un programa en el método **`main`** de la clase **`WrapperClassesUse`** en el que se definan variables para los tipos básicos **`Integer, Double y Character`**. 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 esos 3 tipos envoltorio y asígnales su correspondiente valor de tipo básico.

```
public class WrapperClassesUse {
    public static void main(String[] args) {
        // Assignment of wrapper variables to elementary types
        int i = new Integer(123456);
        // TO COMPLETE ..
        // Writing elementary variables
        System.out.println("int i = " + i);
        // TO COMPLETE ...
        // Assignment of elementary values to wrapper variables
        Integer eI = 123456;
        // TO COMPLETE ...
        // Writing wrapper variables
        System.out.println("Integer I = " + eI);
        // TO COMPLETE ...
    }
}
```

T. Primitivo	C. Envoltorio
int	**Integer**
long	**Long**
short	**Short**
byte	**Byte**
char	**Character**
boolean	**Boolean**
float	**Float**
double	**Double**

2. Clases genéricas predefinidas:

Ejemplo de uso de `ArrayList<E>`

Ejercicio 3: Completa el código en la clase **`ArrayListUse`** para que lea líneas de un fichero y las muestre ordenadas alfabéticamente. En su método **`main`** realiza los siguientes pasos:

1. Crea una instancia de la clase **`ArrayList<E>`** con el tipo puro **`String`** y referenciala con la variable **`list`** del mismo tipo.

2. La lectura se hará con un bucle hasta llegar al final del fichero. En cada iteración, lee una línea del texto, aplicando el método **`nextLine`**() sobre el objeto **`file`**, y añade la línea al objeto de tipo **`ArrayList<String>`** (para ello, pásala como argumento al método **`add(E e)`** aplicado a **`list`**).

3. Ordena las líneas de la lista con el método estático **`sort(List<T> list)`** de la clase **java.util.Collections**. Este metodo recibe como parametro objetos cuya clase implemente el interfaz **`List<E>`**. Entre estas clases se encuentra la clase **`ArrayList<E>`**.

4. Escribe las cadenas de caracteres guardadas en **`list`** invocando el método **`toString`** que por defecto esta definido en la clase **`ArrayList`**.

1. Crea variable **`list`** instancia de **`ArrayList<String>`**

```
public class ArrayListUse {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.print("You must specify an argument: filename");
            System.exit(0);
        }

        File fd = new File(args[0]);
        Scanner file = null;
        // Creating ArrayList object
        // TO COMPLETE ...
        try {
            file = new Scanner(fd);
        }
        catch (FileNotFoundException e) {
            System.err.println("File does not exists " + e.getMessage());
            System.exit(0);
        }
        // Reading file, adding lines to the list
        while (file.hasNext()) {
            // TO COMPLETE ...
        }
        file.close();

        // Sorting the list, writing it to console
        // TO COMPLETE ...
    }
}
```

2. aplicar **`nextLine`**() sobre **`file`**, y añade a **`list`**

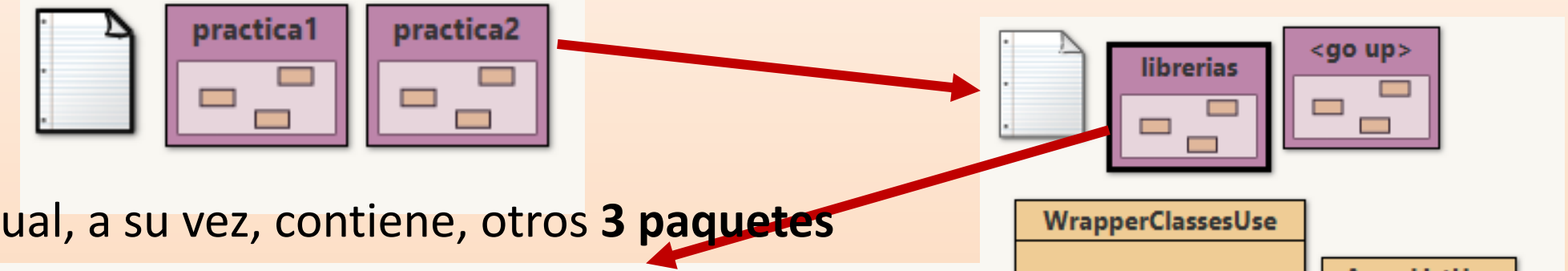
3. Invocar **`sort`** con la variable **`list`**. La lista querará ordenada

4. Visualizar **`list`**

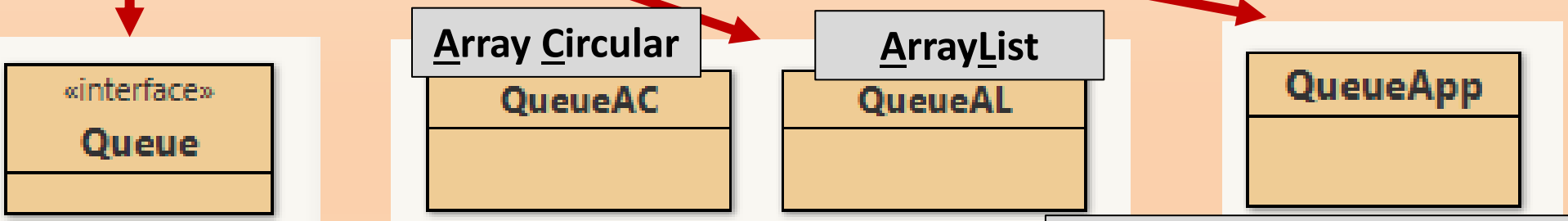
3. Implementación del tipo genérico `Queue<T>`

paquete **librerias**

Si hemos creado el paquete **practica2**, tal y como se explicaba en el ejercicio 1, podemos ver que este paquete, a su vez, contiene el paquete **librerías.**



El cual, a su vez, contiene, otros **3 paquetes**



Contiene la definición de la **interfaz** genérica **`Queue<T>`**

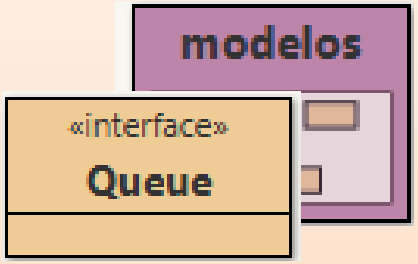
Contiene dos implementaciones diferentes de la interfaz **`Queue<T>`**

Contiene un **main** para crear colas y manipularlas (bien con **`QueueAC<T>`** o **`QueueAL<T>`**)

Interfaces en Java y TAD's: interfaz `Queue<T>`

El paquete **librerias.modelos** contiene la definición de **`Queue<T>`** que especifica las operaciones de una cola genérica. Y, si inspeccionamos su cabecera, podemos ver que se trata de una **interfaz**:

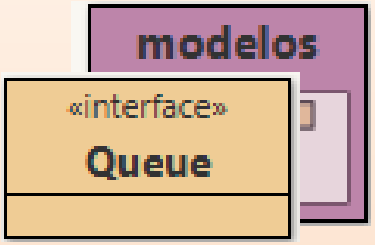
```
public interface Queue<T> {…
```



- Podríamos decir que una interfaz es una **clase totalmente abstracta** (donde todos sus métodos son abstractos).
- Las interfaces introducen cierta flexibilidad en la herencia de Java, y con ello **incrementan la capacidad del polimorfismo en el lenguaje**.
- Las interfaces son clases que se usan para especificar **TAD**'s (Tipos Abstractos de Datos). Al implementar una interfaz, están obligadas ellas y/o sus derivadas a implementar los métodos abstractos heredados de la interfaz.
- El uso de TAD's da lugar a **programas mas robustos y menos propensos a errores**.

Interfaces en Java y TAD's: interfaz `Queue<T>`

Una cola es una estructura lineal **FIFO** (**F**irst **I**n, **F**irst **O**ut) en la que el primer elemento que entra es el primero que sale.



El paquete **librerias.modelos** contiene la definición de la **interfaz `Queue<T>`** que especifica las operaciones de una cola genérica:

```
public interface Queue<T> {

    void enqueue(T e);

    T dequeue();

    int size();

    T first();

    boolean isEmpty();
}
```

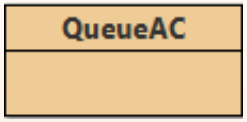
Añade al ***final*** de la cola

Saca la ***cabeza*** de la cola y la devuelve

Devuelve el número de elementos

Devuelve la ***cabeza*** de la cola pero no la saca

Devuelve `true` si la cola es vacía



Implementación con Array Circular: QueueAC<T>

Ejercicio 5: Completa la clase **QueueAC<T>** implementando los métodos del interfaz, teniendo en cuenta la declaración de atributos y la gestión circular del array. Comprueba tu código ejecutando la clase **QueueApp** en la librería aplicaciones.

```
import practica2.librerias.modelos.Queue;

public class QueueAC<T> implements Queue<T> {
    private T[] theArray;
    private int first;
    private int last;
    private int size;
    private static final int MAX = 50;
    public QueueAC() {
        theArray = (T[]) (new Object [MAX]);
        size = 0; first = 0; last = -1;
    }

    public void enqueue(T e) {  completar  }
    public T dequeue() { completar }
    public int size() { completar }
    public T first() { completar }
    public boolean isEmpty() {  completar }

    /** Private method for expanding the array when it is necessar
    private void expandArray() {  ........  }
    /** Private method for increasing the array indexes   */
    private int increase(int i) {
        return (i + 1) % theArray.length;
    }
    public String toString() { ......... }
}
```

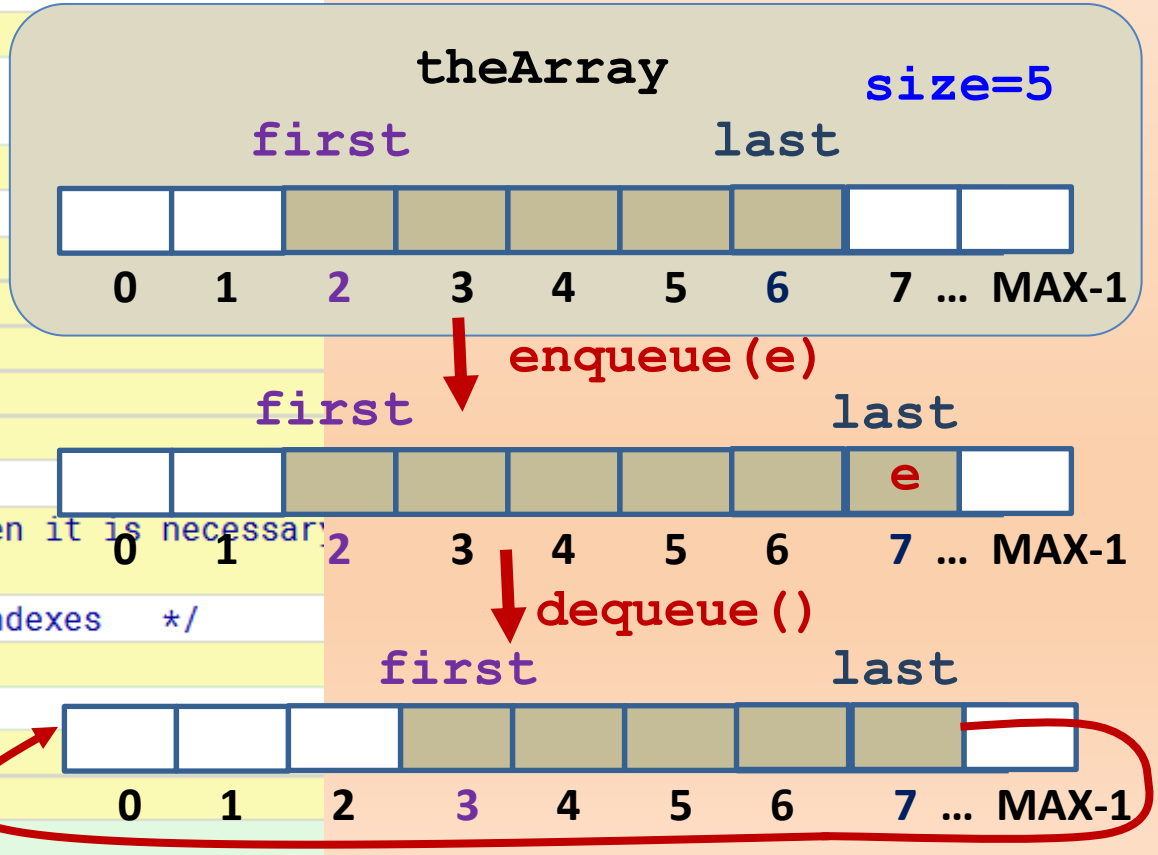
implements Indica que **QueueAC<T>** va a implementar la interfaz **Queue<T>**

Define el array utilizando la variable genérica **T**

Es la posición de la ***cabeza*** de la cola en **theArray**

Es la posición elemento ***final*** de la cola en **theArray**

Contiene el número de elementos de la cola



Prueba de `QueueAC<T>`: Clase `QueueApp`

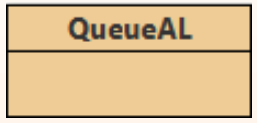
La clase **`QueueApp`** contiene un método **`main`** para probar las implementaciones de **`Queue<T>`**:

```
import practica2.librerias.implementaciones.*;
import practica2.librerias.modelos.*;

public class QueueApp {
    public static void main(String[] args) {
        Queue<Integer> c = new QueueAC<Integer>();
        for (int i = 1; i <= 15; i++) {
            c.enqueue(i);
        }
        System.out.println("Initial queue:\n" + c);
```

Podemos asignar a la variable **`c`** cualquier objeto perteneciente a una clase que implemente **`Queue<Integer>`**

La variable **`c`** es una variable polimórfica que tiene como tipo una interfaz, en este caso **`Queue<Integer>`**



Implementación con ArrayList: QueueAL<T>

Ejercicio 6: Completa la clase **QueueAL<T>** implementando los métodos del interfaz, teniendo en cuenta la declaración de atributos que obliga a usar las operaciones especificadas en la API de la clase **ArrayList**. Comprueba tu código, modificando primero (para poder utilizar la nueva implementación) y ejecutando después la clase **QueueApp** en la librería aplicaciones.

Se debe importar la definición de la interfaz **Queue<T>** y **ArrayList<T>**

QueueAL<T> es una nueva implementación de **Queue<T>**

Solo se necesita un atributo para representar la cola, que se instancia como **ArrayList<T>**

Y se le asigna un nuevo objeto **ArrayList<T>** vacío

```
import practica2.librerias.modelos.Queue;
import java.util.ArrayList;

public class QueueAL<T> implements Queue<T> {

    ArrayList<T> theArray;

    public QueueAL() {
        theArray = new ArrayList<T>();
    }

    public void enqueue(T e) { completar }
    public T dequeue() { completar }
    public int size() { completar }
    public T first() { completar }
    public boolean isEmpty() { completar }

    public String toString() { ..... }
}
```

API ArrayList<E>

```
boolean add(E element)
int size()
E get(int index)
E remove(int index)
boolean isEmpty()
int size()…
```

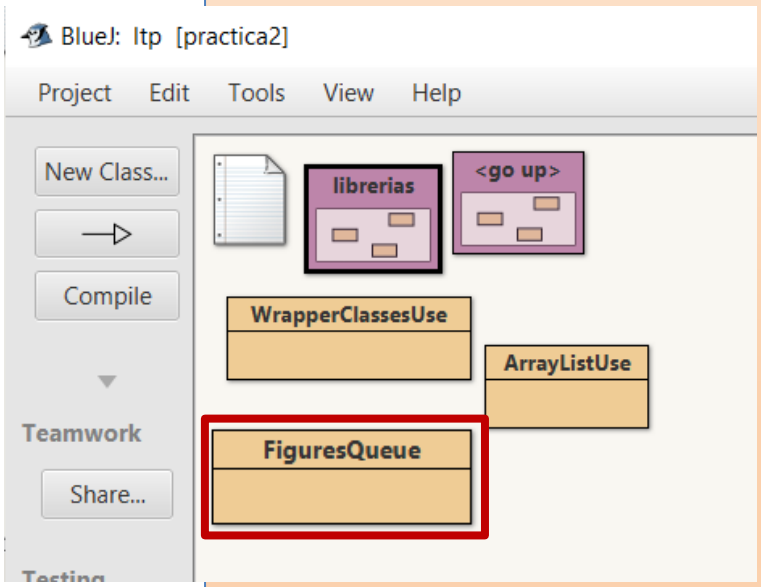
3. Uso de Queue<T> con restricción a Figure

Utilizando **QueueAL<T>** se puede implementar muy fácilmente la clase **FiguresQueue** que representa una cola de figuras. Una implementación básica será la siguiente (**Nota:** ver la definición de la clase en el paquete **practica2**):

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

Ejercicio 7: Teniendo en cuenta esta implementación, identifica en el siguiente programa las líneas que darán 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));
 }
}
```



Calcular el área de una cola de figuras

Ejercicio 8: Añade al paquete **practica2** la clase **`FiguresQueue`**, disponible en Poliformat. Modifica su implementación para que se pueda obtener la suma de las áreas de todas las figuras en la cola (es decir, en el objeto **`this`** invocador) mediante un método de perfil:

```
public double area()
```