WRAPPERS (Clases Envoltorio)

> *"Hijo mío, La felicidad está hecha de pequeñas cosas: Un pequeño yate, una pequeña mansión, una pequeña fortuna, … "*
>
> *Groucho Marx*

Introducción

Las clases wrapper en la API de Java tienen un doble propósito:

1. Proporcionar un mecanismo para **envolver** los valores primitivos en clases, de tal manera que podamos trabajar con valores primitivos tal y como sabemos trabajar con objetos, aprovechando, de esta forma, las características de la POO. Además, como veremos más adelante, podremos utilizar los wrappers en Collections o tipos de datos dinámicas.
2. Proporcionar un conjunto de funciones de utilidad para trabajar con los tipos primitivos.

Listado de los Wrappers disponibles en la API de Java:

Tipo primitivo	Clase Wrapper	Argumentos para el constructor
boolean	Boolean	boolean o String
byte	Byte	byte o String
char	Character	char
double	Double	double o String
float	Float	float
int	Integer	int o String
Long	Long	long o String
short	Short	short o String

Creación de Wrappers

Como se puede observar podemos obtener una instancia de un wrapper a portando o un valor primitivo o un String. Esto se cumple para todos los wrappers excepto para Character que acepta un char. Es importante hacer notar que los constructores que aceptan un String pueden lanzar un *NumberFormatException*.

Ejemplos:

```
// Creación de objetos Byte
Byte b1=new Byte((byte)7);
Byte b2=new Byte("3");
// ------------------
// Creación de objetos Boolean
// true
Boolean bo1t=new Boolean("true"); // Mediante una cadena de caracteres
Boolean bo2t=new Boolean(true); // Mediante la palabra reservada true
Boolean bo3t=new Boolean(Boolean.TRUE); // Mediante el acceso a la constante (static) del propio wrapper Boolean
// false
Boolean bo1f=new Boolean("false"); // Mediante una cadena de caracteres
Boolean bo2f=new Boolean(false); // Mediante la palabra reservada false
Boolean bo3f=new Boolean(Boolean.FALSE); // Mediante el acceso a la constante (static) del propio wrapper Boolean

// Estos objetos se pueden usar en una condición
if (bo1t) {
    // Acciones si bo1t es true
}
```

```
// Creación de objetos Character
Character c1=new Character('J');
Character c2=new Character('j');
// ------------------
// Creación de objetos Double
Double d1=new Double(8.9);
Double d2=new Double("0.5");
Double d3=new Double("3.1a"); // Lanzará una NumberFormatException
// ------------------
// Creación de objetos Float
Float f1=new Float(8.9f);
Float f2=new Float("0.5f");
Float f3=new Float("0.5kg"); // Lanzará una NumberFormatException
// ------------------
// Creación de objetos Integer
Integer i1=new Integer(123);
Integer i2=new Integer("123");
Integer i3=new Integer("1A3"); // Lanzará una NumberFormatException
```

Cuando se crea un objeto wrapper se debe aportar un valor en su constructor, este valor NO se puede modificar a lo largo del tiempo de vida del objeto.

Otra forma de crear un wrapper es mediante el método *valueOf*, el cual se encuentra sobrecargado. El método *valueOf*, es un método estático, por tanto se usa a través del nombre de la clase y devuelve una instancia del wrapper utilizado. Por ejemplo en el caso de *Integer* podemos usar *valueOf* de la siguiente manera:

```
Integer i4=Integer.valueOf(123); // Mediante un tipo primitivo int
Integer i5=Integer.valueOf("123"); // Mediante una cadena de caracteres
Integer i6=Integer.valueOf("10110110", 2);
// Mediante una cadena de caracteres y la base
// en la que se debe interpretar el primer argumento.
// En este ejemplo una cadena de caracteres que representa un num binario y la base 2
// En la creación de i5 e i6 se puede lanzar una NumberFormatException

Float f4=Float.valueOf(9.8f);
Float f5=Float.valueOf("9.8f");
```

La siguiente también es una forma válida de crear wrappers:

```
Double d4=7.9; // El compilador añade new Double(7.9)
Integer i7=1492; // el compilador añade new Integer(1492)
```

xxxValue()

Todos los tipos primitivos numéricos tienen métodos *xxxValue()*, de tal manera que podemos obtener el valor primitivo de un objeto wrapper. Por ejemplo, en el caso de *Integer*:

```
int i7_int=i7.intValue();
float i7_float=i7.floatValue();
double i7_double=i7.doubleValue();
short i7_short=i7.shortValue();
long i7_long=i7.longValue();
byte i7_byte=i7.byteValue();
//  Observar que se devuelve un tipo primitivo.
// Se invoca a través de una instancia de objeto wrapper
```

parseXxx()

Los métodos *parseXxx()* son equivalentes a los *xxxValue()*, es decir, devuelven un tipo primitivo, y son sólo válidos para wrappers numéricos. A diferencia de los métodos *xxxValue()*, son métodos estáticos, por lo que se invocan a través del nombre de la clase. Se les debe pasar por argumento la cadena de caracteres de la que se quiere obtener el tipo primitivo.

```
float f6=Float.parseFloat("5.667f");
float f7=Float.parseFloat("Magic"); // Lanza NumberFormatException
// Notad que es un método estático
```

toString()

Como todo objeto en Java, los wrappers también tienen un método *toString()* heredado de la clase ***Object***. Como podemos imaginar, la funcionalidad de este método es devolver un ***String*** con el valor del valor primitivo del wrapper.

```
Character c3=new Character('j');
String valorDeC3=c3.toString();
System.out.println(valorDeC3);
System.out.println(c3);
// Los dos syso anteriores muestran por la salida estándar el caracter j.
// En el segundo syso toString es invocado implicitamente.
```

equals()

La comparación de 2 objetos wrappers se debe llevar a cabo mediante el método *equals()*. Este método se hereda de **Object** y se encuentra redefinido con la funcionalidad correcta en cada uno de los wrappers. El método *equals* devuelve el valor booleano **true** si los 2 wrappers comparados son iguales y, **false** si los 2 objetos wrappers comparados no son iguales.

```
Integer i8=new Integer(678);
Integer i9=new Integer("678");
if (i8.equals(i9)) {
    System.out.println("i8 e i9 son iguales.");
} else {
    System.out.println("i8 e i9 son difetentes.");
}
```

La comparación de objetos directamente, como ya sabemos, no funciona, ya que estaremos comparando las direcciones de memoria de los objetos.

Bibliografía

https://docs.oracle.com/javase/8/docs/api/index.html?java/lang/package-tree.html

EJERCICIO:

Tomando como base los algoritmos realizados para la *ordenación por burbuja* y la *búsqueda binaria* sobre *Arrays*, adaptarlos para que trabajen con el **wrapper Integer** en lugar de con el **tipo primitivo int**.

Añadir, si no la tenías ya, una función que muestre por consola el contenido del Array. Utilizala para mostrar el contenido del *Array* antes y después de la ordenación.