

# DESARROLLO DE APLICACIONES WEB

## Anexo II - Unidad 8

### W r a p p e r

*1r DAW*

*IES La Mola de Novelda*

*Departament d'informàtica*

# Índice

1.- Envoltura de tipos.....	3
2.- Character.....	4
3.- Boolean.....	4
4.- Envolturas de tipos numéricos.....	4
4.1.- Autoboxing i Auto-UnBoxing.....	6
4.1.1.- Autoboxing y métodos.....	6
4.1.2.- Autoboxing en expresiones.....	7
4.1.3.- Autoboxing en valores booleanos y caracteres.....	8
4.1.4.- Autoboxing y la prevención de errores.....	9
5.- Métodos.....	10

## Anexo Unidad 8: Wrapper

### 1.- ENVOLTURA DE TIPOS

Los tipos de datos se subdividen en dos:

- Primitivos: son los únicos elementos de todo el lenguaje que no son considerados como objetos (y por tanto, no tienen métodos). Como ejemplo tenemos el **int** i el **double**.
- No primitivos: al no ser tipos primitivos, son considerados como objetos (y por tanto, tienen métodos).

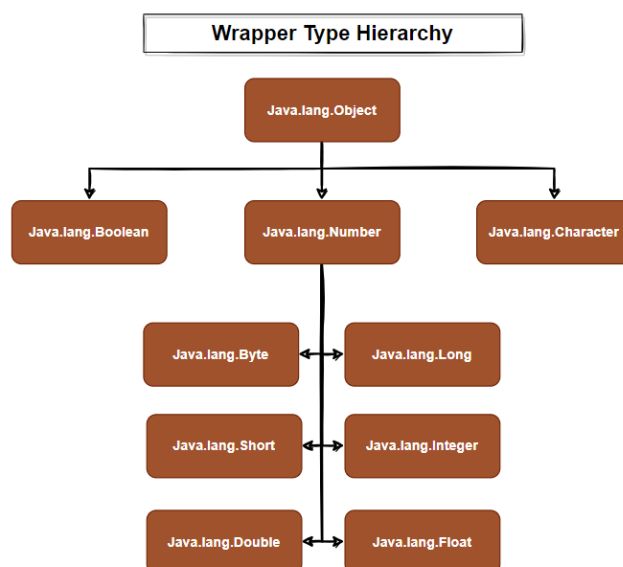
Los tipos primitivos son utilizados para favorecer el rendimiento. Utilizar objetos para valores primitivos agregaría una sobrecarga, incluso para cálculos simples, poco deseable. A pesar de los beneficios de rendimiento ofrecidos por los tipos primitivos, existen ocasiones en que se requiere su representación como un objeto. Por ejemplo, no es posible pasar como parámetro a un método un tipo primitivo por referencia. Además, muchas de las estructuras de datos estándares implementadas por Java trabajan sobre objetos (por ejemplo las **colecciones**), lo que significa que no es posible usar estas estructuras de datos para almacenar datos primitivos.

Para gestionar estas situaciones (y otras) Java provee la envoltura de tipos, que consiste en proporcionar clases que encapsulan a un tipo primitivo dentro de un objeto.

Las envolturas de tipos son Double, Float, Long, Integer, Short, Byte, Character y Boolean. Estas clases ofrecen un conjunto amplio de métodos que permiten integrar completamente a los tipos primitivos dentro de la jerarquía de objetos de Java.

Si nos metemos en la API de Java, y buscamos el *package java.lang*, podemos ver que nos aparecen los distintos tipos de *Wrappers* en el apartado *Class Summary*.

Todos los Wrappers a su vez dependen de *java.lang.Object* ya que, como hemos dicho, los Wrappers no dejan de ser objetos y por tanto descienden de la clase *Object* (todos los objetos descienden de la clase *Object*).



## 2.- CHARACTER

*Character* es la envoltura del tipo *char*. El constructor para *Character* es:

***Character(char ch)***

Donde **ch** especifica el carácter que será envuelto por el objeto *Character* que está siendo creado. Para obtener el valor **char** contenido en el objeto *Character*, se llama al método *charValue()*, como se muestra a continuación el cuál devolverá al carácter encapsulado.:

***char charValue()***

Ejemplo: *EnvoltoriCharacter.java*

## 3.- BOOLEAN

*Boolean* es la envoltura de los valores del tipo primitivo boolean. El cual define estos constructores:

***Boolean(boolean boolValue)***

***Boolean(String boolString)***

En la primera versión, **boolValue** debe ser *true* o *false*. En la segunda versión, si **boolString** contiene la cadena "*true*" (en minúsculas o mayúsculas), entonces el nuevo objeto **Boolean** será verdadero, de otra forma, será falso.

Para obtener el valor del objeto *Boolean*, se utiliza el método ***booleanValue()***, como se muestra a continuación, el cual devolverá el valor de tipo *boolean* equivalente al del objeto invocado:

***boolean booleanValue()***

Ejemplo: *EnvoltoriBoolean.java*

## 4.- ENVOLTURAS DE TIPOS NUMÉRICOS

Las envolturas **Byte**, **Short**, **Integer**, **Long**, **Float** y **Double** son las más comúnmente usadas. Todas ellas heredan de la clase abstracta **Number**. **Number** declara métodos que regresan el valor de un objeto en cada uno de los diferentes formatos. Estos métodos se muestran a continuación:

***byte byteValue()***

***double doubleValue()***

*float floatValue()*

*int intValue()*

*long longValue()*

*short shortValue()*

Ejemplo: ***doubleValue()*** regresa el valor de un objeto como un valor de tipo *double*, ***floatValue()*** regresa el valor como un valor de tipo *float*, y así sucesivamente

Todas las envolturas de tipos numéricos definen constructores que permiten a un objeto ser construido a partir de un valor dado o a partir de una cadena que represente el valor.

Ejemplo: Los constructores definidos para la clase ***Integer*** son:

***Integer(int num)***

***Integer(String str)***

Si ***str*** no contiene un valor numérico válido entonces una excepción de tipo ***NumberFormatException*** es lanzada. Todas las envolturas de tipo sobrescriben al método ***toString()***, el cual regresa en una forma compresible el valor contenido dentro de la envoltura. Esto permite, por ejemplo, desplegar el valor del objeto envuelto cuando es usado en un ***println()*** sin tener que convertirlo a su tipo primitivo.

Ejemplo: Este programa envuelve el valor entero de 100 dentro de un objeto ***Integer*** llamado ***iOb***. El programa entonces obtiene ese valor llamando ***intValue()*** y almacena el resultado en ***i***. El proceso de encapsulación de un valor dentro de un objeto es llamado ***boxing***. El proceso de extracción del valor desde una envoltura de tipos es llamado ***unboxing***.

// demostración de envoltura de tipos

```
class Wrap {  
    public static void main(String args[]) {  
        Integer iOb = new Integer(100); // BOXING  
        int i = iOb.intValue(); // UNBOXING  
        System.out.println(i + " " + iOb); // muestra 100 100  
    }  
}
```

El mismo procedimiento general utilizado por el programa anterior para ***boxing*** y ***unboxing*** ha sido empleado desde la versión original de Java. Sin embargo, con la llegada del JDK 5, Java mejoró considerablemente esta característica adicionando el concepto de ***autoboxing*** que se describe a continuación.

## 4.1.-AUTOBOXING I AUTO-UNBOXING

A partir de JDK 5, Java agregó dos importantes características: **autoboxing** y **auto-unboxing**.

- **Autoboxing** es el proceso por medio del cual un tipo primitivo es automáticamente encapsulado dentro de un objeto generado por envoltura de tipos, en cualquier lugar donde un objeto de ese tipo se requiera. No es necesario construir explícitamente un objeto. Con el **autoboxing** ya no se necesita construir manualmente un objeto para envolver un tipo primitivo. Sólo se necesita asignar el valor a una referencia de una envoltura del tipo. Java automáticamente construye el objeto sin tener que crearlo explícitamente con **new**.

```
Integer iOb = 100; // autoboxing de un valor de tipo int.
```

- **Auto-unboxing** es el proceso mediante el cual el valor de un objeto (generado por envoltura de tipos) es automáticamente despojado de su envoltura de tipo cuando el valor es requerido. No es necesario llamar a un método tal como **intValue( )** o **doubleValue( )**. Simplemente debemos asignar el objeto referenciado a una variable de tipo primitivo ya que Java gestiona los detalles automáticamente.

```
int i = iOb; //auto-unboxing
```

Ejemplo: Nueva versión del ejemplo anterior però rescrito utilizando autoboxing /unboxing

```
// Ejemplo de autoboxing / auto-unboxing
class AutoBox {
    public static void main (String args []) {
        Integer iOb = 100; // autoboxing un valor de tipo int
        int i = iOb; // auto-unboxing
        System.out.println(i+ " " + iOb); // muestra 100 100
    }
}
```

### 4.1.1.- Autoboxing y métodos

Además de ocurrir en los casos simples de asignación de valores, el **autoboxing** ocurre en cualquier momento que un tipo primitivo debe ser convertido en un objeto y **auto-unboxing** toma lugar cuando un objeto debe ser convertido a un tipo primitivo. Así, **autoboxing** y **auto-unboxing** pueden ocurrir cuando un argumento se pasa a un método, o cuando un valor es devuelto por un método. Ejemplo:

```
// autoboxing y auto-unboxing ocurren cuando
```

```
// un método recibe argumentos o devuelve valores
class AutoBox2 {
    // este método recibe un argumento del tipo Integer y regresa un valor del tipo primitivo int
    static int m (Integer v) {
        return v ; // auto-unboxing el objeto v a un valor int
    }
    public static void main(String args[]) {
        // Se envía un valor de tipo int al método m() y asigna el valor a un objeto Integer.
        // El argumento 100 sufre autoboxing, al igual que el valor regresado por el método
        Integer iOb = m(100);
        System.out.println(iOb);
    }
}
```

#### 4.1.2.- Autoboxing en expresiones

En general, autoboxing y auto-unboxing ocurren en cualquier momento en que una conversión de un valor a un objeto o de un objeto a un valor es requerida. Esto se aplica también a las expresiones. Ejemplo:

// autoboxing y auto-unboxing ocurren en las expresiones.

```
class AutoBox3 {
    public static void main(String args[]) {
        Integer iOb, iOb2;
        int i;
        iOb = 100;
        System.out.println("Valor original de iOb: " + iOb);
        // El código siguiente aplica automáticamente unboxing a iOb,
        // realiza un incremento y luego aplica autoboxing nuevamente
        // para colocar el resultado en iOb
        ++iOb;
        System.out.println("Después de ++iOb: " + iOb);
        // La expresión se evalúa después de que a iOb se le aplica unboxing,
        // al resultado se le aplica autoboxing y luego se almacena en iOb2.
        iOb2 = iOb + (iOb / 3);
        System.out.println("iOb2 después de evaluar la expresión es: " + iOb2);
        // La misma expresión se evalúa ahora sin que sea necesario
        // aplicar autoboxing al resultado
        i = iOb + (iOb / 3);
        System.out.println("i después de evaluar la expresión es: " + i);
    }
}
```

```
Valor original de iOb: 100
Después de ++iOb: 101
iOb2 después de evaluar la expresión es: 134
i después de evaluar la expresión es: 134
```

El proceso de **auto-unboxing** también permite que se mezclen diferentes tipos de objetos numéricos en una expresión. Ejemplo:

```
class AutoBox4 {  
    public static void main(String args[]){  
        // autoboxing y auto-unboxing dentro de expresiones  
        Integer iOb = 100;  
        Double dOb = 98.6;  
        dOb = dOb + iOb;  
        System.out.println("dOb después de la expresión: " + dOb);  
    }  
}
```

dOb después de la expresión: 198.6

Como se puede ver, tanto el objeto **Double dOb** como el objeto **Integer iOb** participan en la adición y el resultado pasa por **autoboxing** antes de ser almacenado en **dOb**.

Debido al **auto-unboxing**, es posible utilizar objetos numéricos enteros para controlar una sentencia **switch**. Ejemplo:

```
class AutoBox4 {  
    public static void main(String args[]){  
        Integer iOb = 2;  
        switch (iOb) {  
            case 1: System.out.println ("uno");  
                break;  
            case 2: System.out.println ("dos");  
                break;  
            default: System.out.println("error");  
        }  
    }  
}
```

Cuando la expresión en el **switch** es evaluada, a **iOb** se le aplica **unboxing** y su valor entero es obtenido.

Los ejemplos muestran cómo la aplicación de **autoboxing** y **auto-unboxing** a objetos numéricos dentro de expresiones es intuitiva y fácil. En el pasado, un código similar habría involucrado conversión de tipos y llamadas a métodos, como por ejemplo **intValue()**.

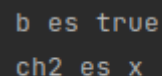
#### 4.1.3.- Autoboxing en valores booleanos y caracteres

Los procesos de **autoboxing** y **auto-unboxing** se aplican también las envoltura de **Boolean** y **Character**. Ejemplo:



// Autoboxing y unboxing de objetos Boolean y Character.

```
class AutoBox5 {  
    public static void main(String args[]) {  
        // autoboxing y unboxing aplicado a un valor boolean.  
        Boolean b = true;  
        // b pasa por auto-unboxing cuando es utilizada en una expresión condicional  
        if(b)  
            System.out.println("b es true");  
        // autoboxing y unboxing aplicado a un valor char.  
        Character ch = 'x'; // autoboxing un char  
        char ch2 = ch; // unboxing un char  
        System.out.println("ch2 es " + ch2);  
    }  
}
```



```
b es true  
ch2 es x
```

Con el **auto-unboxing**, un objeto de tipo **Boolean** también puede ser utilizado para controlar cualquiera de las sentencias de ciclo de Java. Cuando un objeto **Boolean** es utilizado en la expresión condicional de un **while**, **for** o **do/while**, se le aplica automáticamente **unboxing** para convertirlo en su equivalente **boolean**.

#### 4.1.4.- Autoboxing y la prevención de errores

Además de las facilidades que ofrecen, también ayudan a prevenir errores. Ejemplo:

// Aquí se produce un error debido al unboxing manual

```
class UnboxingError {  
    public static void main(String args []) {  
        Integer iOb = 1000; // autoboxing del valor 1000  
        int i = iOb.byteValue(); // ¡unboxing manual como tipo byte!  
        System.out.println(i); // ¡esto NO desplegará 1000!  
    }  
}
```



```
-24
```

Vemos que **iOb** pasa por un **unboxing** manual por la llamada al método **byteValue()**, el cual produce un truncamiento del valor 1000 almacenado en **iOb**. Esto da como resultado que el valor -24 sea asignado a **i**. El **auto-unboxing** previene este tipo de errores porque el valor en **iOb**, mediante **auto-unboxing**, dará lugar a un valor compatible con **int**.

Comúnmente, **autoboxing** siempre crea el objeto correcto, y **auto-unboxing** siempre produce el valor correcto. No hay forma de que el proceso produzca un tipo de objeto o un valor incorrecto. En general, los nuevos programas deberían utilizar **autoboxing** y **auto-unboxing**. Es la forma en que los programas de Java son escritos.

Podría resultar tentador utilizar objetos tales como **Integer** o **Double** y abandonar a los tipos primitivos del todo. Según el caso, utilizar objetos sería menos eficiente que si utilizamos los tipos primitivos. La razón es que cada aplicación de **autoboxing** y **auto-unboxing** agrega trabajo adicional que no se presenta cuando se usan tipos primitivos. Ejemplo:

```
// uso incorrecto de autoboxing y unboxing
Double a, b, c;
a = 10.0;
b = 4.0;
c = Math.sqrt(a*a + b*b);
System.out.println("La hipotenusa es: " + c);
```

En general, el uso de la envoltura de tipos debe restringirse solamente a los casos en los cuales la representación de un objeto de un tipo primitivo sea requerida. **Autoboxing** y **auto-unboxing** no fueron agregados a Java para eliminar los tipos primitivos.

## 5.- MÉTODOS

Los envoltorios, como objetos que son, disponen de una serie de métodos. Los más útiles son los que interpretan cadenas de caracteres (a menudo leídas del teclado) que representan valores y los convierten. Por ejemplo, la clase **Double** dispone del método:

```
static double parseDouble(String cadena)
```

A este método se le pasa una cadena que representa un número real y lo convierte en un **double**. El resto de los **wrappers** disponen de métodos análogos. Ejemplo:

```
String cad = "23.546";
double t = Double.parseDouble(cad); //La variable real t contiene el valor decimal 23.546
System.out.println(t);
```

Aunque los métodos a utilizar los podemos encontrar en la API de Java, todas estas clases tienen los métodos **parseXxx** (siendo **Xxx** el tipo de datos al que se quiere parsear el **String**) y el método **toString** para convertir un valor del tipo que representan a cadena de caracteres. Veamos algunos ejemplos:

```
// --- operaciones con el tipo int ---  
int i = 43;  
// convierto de int a String  
String sInt = Integer.toString(i);  
// convierto de String a int  
int i2 = Integer.parseInt(sInt);  
// --- operaciones con el tipo double ---  
double d = 24.2;  
// convierto de double a String  
String sDouble = Double.toString(d);  
// convierto de String a double  
double d2 = Double.parseDouble(sDouble);
```