CLASES ENVOLTORIO (Wrappers)

Para cada uno de los tipos de datos básicos, Java proporciona una clase que lo representa. A estas clases se las conoce como clases envoltorio, y sirven para dos propósitos principales:

- Encapsular un dato básico en un objeto, es decir, proporcionar un mecanismo para "envolver" valores primitivos en un Objeto para que los primitivos puedan ser incluidos en actividades reservadas para los objetos, como ser añadido a una colección.
- Proporcionar un conjunto de funciones útiles para los primitivos. La mayoría de estas funciones están relacionadas con varias conversiones: convirtiendo primitivos a String y viceversa y convirtiendo primitivos y objetos String en diferentes bases, tales como, binario, octal y hexadecimal.

Hay una clase de envoltorio para cada primitivo en Java. Por ejemplo, la clase de envoltura para `int` es `Integer`, la clase `Float` es de `float` y así todas. Recuerda que los nombres de los datos primitivos es simplemente el nombre en minuscula, a excepción de `int` que es `Integer` y `char` que es `Character`.
Todas las clases envoltorio están definidas en el paquete por defecto *java.lang*.

La siguiente tabla nos muestra la lista de clases envoltorio en la API de Java:

Primitivo	Clase envoltorio	Argumentos del constructor
byte	Byte	byte o String
short	Short	short o String
int	Integer	int o String
long	Long	long o String
float	Float	float, double o String
double	Double	double o String
boolean	Boolean	boolean o String
char	Character	char

Encapsulamiento de un tipo básico. Construcción de los envoltorios.
Todas las clases envoltorio excepto Character provee dos constructores: uno que toma un dato primitivo y otro que toma una representacion String del tipo que esta siendo construido. Por ejemplo:

```
Integer i1 = new Integer(42);
Integer i2 = new Integer("42");
```

```
Float f1 = new Float(3.14f);
Float f2 = new Float("3.14f");
```

La clase Carácter suministra sólo un constructor que toma un char como argumento, ejemplo:

```
Character c1 = new Character('c');
```

Los constructores para los envoltorios booleanos toman también un valor booleano true o false, una cadena sensible a mayúsculas con el valor true o false. Hasta
Java 5, un objeto boolenao no podía ser usado como una expresión en un test booleano, por ejemplo:

```
Boolean b = new Boolean("false");
if (b) // no compila en Java 1.4 o anterior
```

En Java 5, un objeto booleano puede ser usado en el test, porque el compilador automaticamente desempacará (unbox) el Boolean a booleano.

El problema con el que nos encontramos cada vez más usualmente es que desde la versión 9 del JDK, esta forma envolver automáticamente un tipo primitivo está marcada como deprecated, es decir, está obsoleta y va a dejar de estar disponible en algún momento. Actualmente, tal y como nos recomienda el IDE desde la propia advertencia, es aconsejable el utilizar el método `valueOf()` *para introducir un valor dentro de un objeto wrapper.*

El método valueOf()
El método estatico `valueOf()` es suministrado en la mayoría de las clases de envoltorio para darte la capacidad de crear objetos envoltorio. También toma una cadena como representación del tipo de Java como primero argumento. Este método toma un argumento adicional, int radix, que indica la base (por ejemplo, binario, octal, o hexadecimal) del primer argumento suministrado, por ejemplo:

```
Integer i2 = Integer.valueOf("101011", 2); // convierte
101011 a 43 y lo asigna al objeto Integer i2
```

```
Float f2 = Float.valueOf("3.14f"); // asigna 3.14 a f2
```

Usando las herramientas de conversión para envoltorios
Como dijimos antes, la segunda gran función de un envoltorio es la conversión.
Los siguientes métodos son los más usados:

xxxValue()
Los métodos `xxxValue()` se utilizan cuando se necesita convertir el valor de un envoltorio numérico a primitivo. Todos los métodos de esta familia no tienen argumentos. Hay 36 métodos. Cada una de las seis clases envoltorio, tiene seis métodos, así que cualquier número envoltorio puede ser convertido a cualquier tipo primitivo, por ejemplo:

```
Integer i2 = new Integer(42); // Crea nuevo objeto envoltorio
byte b = i2.byteValue(); // convierte el valor de i2 a byte
```

```
short s = i2.shortValue(); // otro método de xxxValue para
números enteros
double d = i2.doubleValue();// otro más
```

```
Float f2 = new Float(3.14f); // crea nuevo objeto envoltorio
short s = f2.shortValue(); // convierte valor de f2 a short
System.out.println(s); // resultado: 3 (truncado, no
redondeado)
```

parseXxx() y valueOf()

Los seis métodos `parseXxx()` (uno por cada tipo de envoltorio) son muy parecidos a los de `valueOf()`. Los dos toman una cadena como argumenta arrojando NumberFormatException (comunmente denominado NFE) si el argumento cadena no esta apropiadamente formateado, y puede convertir objetos de cadena a diferentes bases (radix), cuando el dato primitivo es cualquiera de los cuatro tipos de números enteros. La diferencia entre ambos métodos es:

- `parseXxx()` devuelve el primitivo.
- `valueOf()` devuelve el recién creado objeto envoltorio del tipo invocado en el método.

Aquí se muestran algunos de los ejemplos de estos métodos en acción:

```
double d4 = Double.parseDouble("3.14"); // convierte un
String a un primitivo.

System.out.println("d4 = " + d4); // resultado: d4 = 3.14
```

```

System.out.println(d5 instanceof Double); // resultado: true
Double d5 = Double.valueOf("3.14"); // crea un objeto Double
```

Los siguientes ejemplos de envoltura usan el parámetro radix (en esta caso binario):

```
long L2 = Long.parseLong("101010",2); //cadena binaria a long
System.out.println("L2 = " + L2); // resultado es: L2 = 42
```

```
Long L3 = Long.valueOf("101010", 2); // cadena binaria a Long
System.out.println("L3 value = " + L3); // L3 value = 42
```

toString()

La clase alfa, o sea, la primera clase de Java, Object, tiene un metodo `toString()`.

Asi que todas las clases de Java tienen también este método. La idea de `toString()` es permitirte conseguir una representación mas significativa del objeto. Por ejemplo, si tienes una coleccion de varios tipos de objetos, puedes hacer un bucle a través de la colección e imprimir algunas de las representaciones más significativas de cada objeto usando `toString()`, que está presente en todas las clases. Hablaremos mas de `toString()` en las colecciones en el capitulo correspondiente, ahora vamos a centrarnos en como funciona `toString()` en las clases envoltorio, que como ya sabemos, están marcadas como clases finales (ya que son inmutables). Todas las clases envoltorio devuelven una cadena con el valor del objeto primitivo del objeto, por ejemplo:

```
Double d = new Double("3.14");System.out.println("d = "+
d.toString()); // “d = 3.14”
```

Todas las clases envoltorio suministran un método sobrecargado, que toma un número primitivo de un tipo apropiado (`Double.toString()` toma un double, `Long.toString()` toma un long, etc...) y naturalmente devuelve una cadena.

```
String d = Double.toString(3.14); // d = "3.14"
```

Para terminar, los enteros y los long suminstran un tercer método, es estático. Su primer argumento es el dato primitivo, y el segundo argumento es el radix. El radix le dice al método cómo coger el primer argumento, lo que significa que si el radix es 10 (base 10, por defecto), lo convertirá a la base sumnistrada, y devolverá una cadena, por ejemplo:

```
String s = "hex = "+ Long.toString(254,16); // s = "hex = fe"
```

toXxxString() (Binario, Hexadecimal, Octal)

Las clases envoltorio Long e Integer te permiten convertir números en base 10 a otras bases. Estas conversiones toman un int o un long y devuelven una cadena que representa la conversión del número. Por ejemplo:

```
String s3 = Integer.toHexString(254); // 254 en hexadecimal

System.out.println("254 es " + s3); // resultado: "254 es fe"
```

```
String s4 = Long.toOctalString(254); //convierte 254 a octal

System.out.print("254(oct) = " + s4); // "254(oct) = 376"
```

En esencia, los metodos para la conversión de envoltorios son:

- primitive `xxxValue()`, convierte un envoltorio a primitivo.
- primitive `parseXxx(String)`, convierte una cadena a un primitivo.
- Envoltorio `valueOf(String)`, convierte cadena a envoltorio.

Autoboxing

Con la introducción de autoboxing (y unboxing) en Java 5 muchas de las operaciones de envoltura que realizaban los programadores manualmente ahora se hacen de manera automática .

El autoboxing representa otra de las nuevas características del lenguaje incluidas a partir de la versión Java 5 siendo, probablemente, una de las más prácticas. Consiste en la encapsulación automática de un dato básico en un objeto de envoltorio, mediante la utilización del operador de asignación. Por ejemplo, según se ha explicado anteriormente, para encapsular un dato entero de tipo int en un objeto Integer, deberíamos proceder del siguiente modo:

```
int p = 5;Integer n = new Integer (p);
```

Utilizando el autoboxing la anterior operación puede realizarse de la siguiente forma:

```
int p = 5; Integer n = p;
```

Es decir, la creación del objeto de envoltorio se produce implícitamente al asignar el dato a la variable objeto.

De la misma forma, para obtener el dato básico a partir del objeto de envoltorio no será necesario recurrir al método **`xxxValue()`**, esto se realizará implícitamente al utilizar la variable objeto en una expresión. Por ejemplo, para recuperar el dato encapsulado en el objeto **`n`** utilizaríamos:

```
int a = n;
```

A la operación anterior se le conoce como autounboxing y, al igual que el autoboxing, solamente pueden ser utilizadas a partir de la versión Java 5.

El autoboxing/autounboxing permite al programador despreocuparse de tener que realizar de forma manual el encapsulado de un tipo básico y su posterior recuperación, reduciendo el número de errores por esta causa.

El siguiente método representa un ejemplo más de la utilización del autoboxing / autounboxing, en él se hace uso de esta técnica para mostrar por pantalla los números recibidos en un array de objetos Integer:

```
Integer[] nums = {7, 24, 12, 86, 101};
int suma = 0;
for(Integer n : nums){
      suma += n; //autounboxing
      System.out.println("El número vale " + n);
}
System.out.println("La suma total es " + suma);
```

Hay que tener en cuenta que las operaciones aritméticas habituales (suma, resta, multiplicación...) están definidas solo para los datos primitivos por lo que las clases envoltura no sirven para este fin.

Las variables primitivas tienen mecanismos de reserva y liberación de memoria más eficaces y rápidos que los objetos por lo que deben usarse datos primitivos en lugar de sus correspondientes envolturas siempre que se pueda.

¿Por qué no trabajamos directamente con Wrappers?

Los primitivos han estado presentes desde los orígenes de Java en 1996. A día de hoy los tipos primitivos siguen utilizándose principalmente por el notable rendimiento que ofrecen.

BigInteger y BigDecimal

BigInteger y **BigDecimal** no son clases envoltorio como tal aunque sí son clases contenedoras de los tipos primitivos **int** y **double**, y tienen ventajas sobre los tipos primitivos. Cuando necesites usar números grandes en Java la mejor opción es usar estas clases, ya que su límite de almacenamiento es el límite de memoria que tenga la máquina virtual de Java.

Las dos clases pertenecen al paquete **java.math** y ponen a disposición del desarrollador de múltiples constructores.

Los objetos creados de estas clases son inmutables, es decir, no se puede modificar lo que contienen, en su lugar necesitan ser reasignados.

Además, estas clases proporcionan algunos métodos bastante interesantes, como las operaciones básicas o saber si el número es primo o no. Como siempre vamos con algunos ejemplos de apoyo:

Operaciones básicas con la clase BigInteger

En el siguiente ejemplo se hace uso de los métodos **add()**, **substract()**, **multiply()** y **divide()** para realizar las operaciones básicas de suma, resta, multiplicación y división:

```
import java.math.BigInteger;
public class PruebaOperacionesBasicas {
    public static void main(String args[]){
        BigInteger entero1 = BigInteger.valueOf(45);
        BigInteger entero2 = BigInteger.valueOf(15);

        //sumar con metodo add()
        String texto = "La suma entre " + entero1 +
                " y " + entero2 + " es " + entero1.add(entero2);

        //restar con metodo substract()
        texto += "\nLa resta entre " + entero1 +
                " y " + entero2 + " es "+entero1.subtract(entero2);

        //multiplicar con metodo multiply()
        texto += "\nEl producto de " + entero1 +
                " y " + entero2 + " es "+ entero1.multiply(entero2);

        //dividir con metodo divide()
        texto += "\nLa división de " + entero1 +
                " y "+ entero2 + " entre "+ entero1.divide(entero2);
        System.out.println(texto);
    }
}
```

Números primos en Java

Este es un típico ejercicio que nos plantean en la universidad; se trata de hacer un listado de los primeros X números primos. En este caso, la clase BigInteger nos proporciona dos métodos muy interesantes: **`isProbablePrime()`** y **`nextProbablePrime()`**. El primero de ellos devuelve true si el número es probablemente primo (recordemos que es complejo saber si un número es o no primo cuando es demasiado grande); el segundo devuelve el próximo posible número primo a partir del valor que contenga el objeto.

La siguiente aplicación usa el método **`nextProbablePrime()`** para listar los primeros 2000 números primos:

```
import java.math.BigInteger;
public class PruebaNumerosPrimos {
    public static void main(String[] args){
        // iniciar el entero en cero
        BigInteger entero = BigInteger.ZERO;
        StringBuffer texto = new StringBuffer();
        for(int i = 0; i < 2000; i++){
            entero = entero.nextProbablePrime();
            texto.append(entero + "\n");
        }
        System.out.println(texto);
    }
}
```

Otros métodos interesantes

La clase `BigInteger` proporciona además otros métodos que nos ahorrarán bastante tiempo y líneas de código:

- **`pow()`**, nos permite elevar un número a la potencia que deseemos.
- **`compareTo()`**, nos permite comparar si un BigInteger es mayor, igual o menor que otro.
- **`min`** y **`max()`**, nos permiten saber cuál de los dos BigInteger es menor o mayor.

Como se mencionó al principio, también es posible usar la clase `BigDecimal` que funciona de manera similar pero con números de coma flotante.