# Clase String

Un String en Java representa una **cadena de caracteres no modificable**.

Todos los literales de la forma "*cualquier texto*", es decir, literales entre comillas dobles, que aparecen en un programa java se implementan como objetos de la clase String.

## CREAR UN STRING

Se puede **crear un String** de varias formas, entre ellas:

- Utilizando una **cadena de caracteres** entre comillas:

```
String s1 = "abcdef";
```

- Utilizando **operador de concatenación +**con dos o más objetos String:

```
String s2 = s1 + "ghij";   //s2 contiene "abcdefghij"
String s3 = s1 + s2 + "klm"; //s3 contiene " abcdefabcdefghijklm"
```

Además la clase String proporciona varios **constructores**, entre ellos:

| CONSTRUCTOR | DESCRIPCIÓN |
|---|---|
| String() | Constructor por defecto. El nuevo String toma el valor ""<br>String s = new String();  //crea el string s vacío.<br>Equivale a:  String s = ""; |
| String(String s ) | Crea un nuevo String, copiando el que recibe como parámetro.<br>String s = "hola";<br>String s1 = new String(s);<br>//crea el String s1 y le copia el contenido de s |
| String( char[] v ) | Crea un String y le asigna como valor los caracteres contenidos en el array recibido como parámetro.<br>char [] a = {'a', 'b', 'c', 'd', 'e'};<br>String s = new String(a);<br>//crea String s con valor "abcde" |
| String(char[] v, int pos, int n) | Crea un String y le asigna como valor los n caracteres contenidos en el array recibido como parámetro, a partir de la posición pos.<br>char [] a = {'a', 'b', 'c', 'd', 'e'};<br>String s = new String(a, 1, 3);<br>//crea String s con valor "bcd"; |

## MÉTODOS DE LA CLASE STRING

La clase String proporciona métodos para el tratamiento de las cadenas de caracteres: acceso a caracteres individuales, buscar y extraer una subcadena, copiar cadenas, convertir cadenas a mayúsculas o minúsculas, etc.

| MÉTODO | DESCRIPCIÓN |
|---|---|
| length() | Devuelve la longitud de la cadena |
| indexOf('caracter') | Devuelve la posición de la primera aparición de carácter |
| lastIndexOf('caracter') | Devuelve la posición de la última aparición de carácter |
| charAt(n) | Devuelve el carácter que está en la posición n |
| substring(n1,n2) | Devuelve la subcadena comprendida entre las posiciones n1 y n2-1 |
| toUpperCase() | Devuelve la cadena convertida a mayúsculas |
| toLowerCase() | Devuelve la cadena convertida a minúsculas |
| equals("cad") | Compara dos cadenas y devuelve true si son iguales |
| equalsIgnoreCase("cad") | Igual que equals pero sin considerar mayúsculas y minúsculas |
| compareTo(OtroString) | Devuelve 0 si las dos cadenas son iguales. <0 si la primera es alfabéticamente menor que la segunda ó >0 si la primera es alfabéticamente mayor que la segunda. |
| compareToIgnoreCase(Otro String) | Igual que compareTo pero sin considerar mayúsculas y minúsculas. |
| valueOf(N) | Método estático. Convierte el valor N a String. N puede ser de cualquier tipo. |

Los puedes consultar todos en la API de Java:

http://docs.oracle.com/javase/7/docs/api/index.html?java/lang/String.html

Debemos recordar que:

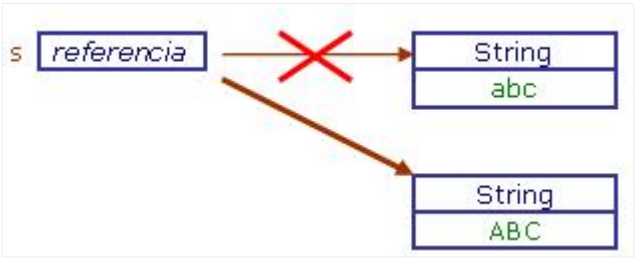
**Los objetos String no son modificables.**

Por lo tanto, los métodos que actúan sobre un String con la intención de modificarlo lo que hacen es crear un nuevo String a partir del original y devolverlo modificado.

Por ejemplo: Una operación como convertir a mayúsculas o minúsculas un String no lo modificará sino que creará y devolverá un nuevo String con el resultado de la operación.

String s = "abc";


s = s.toUpperCase(); //convertir a mayúsculas el contenido del String s




El **recolector de basura** es el encargado de eliminar de forma automática los objetos a los que ya no hace referencia ninguna variable.

## EL OPERADOR DE CONCATENACIÓN +

La clase proporciona el operador + (concatenación) para unir dos o más String.

El resultado de aplicar este operador es un nuevo String concatenación de los otros.

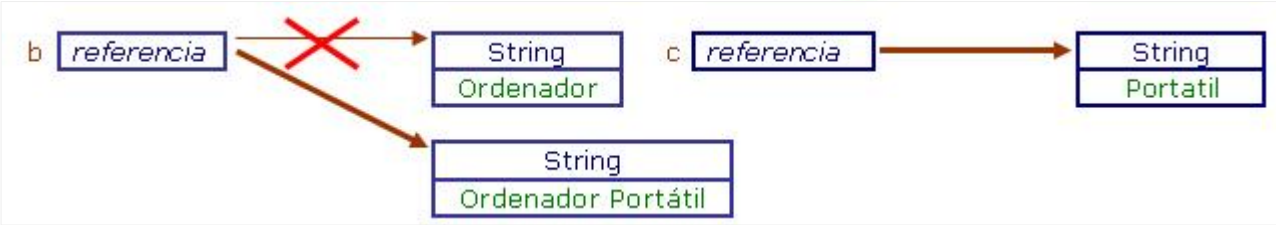
Por ejemplo:

String b = "Ordenador";

String c = " Portátil";


La operación: b = b + c;

crea un nuevo String (b + c) y le asigna su dirección a b:


b
referencia
String
Ordenador
String
Ordenador Portátil
c
referencia
String
Portatil

# Java StringBuilder - StringBuffer

La clase String es una clase no modificable. Esto quiere decir que cuando se modifica un String se crea un nuevo objeto String modificado a partir del original y el recolector de basura es el encargado de eliminar de la memoria el String original.

Java proporciona la clase StringBuffer y a partir de Java 5 la clase StringBuilder para trabajar con cadenas de caracteres sobre las que vamos a realizar modificaciones frecuentes de su contenido.

La **diferencia entre StringBuffer y StringBuilder** es que los métodos de StringBuffer están sincronizados y los de StringBuilder no lo están. Por este motivo StringBuilder ofrece mejor rendimiento que StringBuffer y la utilizaremos cuando la aplicación tenga un solo hilo de ejecución.

En general decidiremos cuando usar String, StringBuilder o StringBuffer según lo siguiente:

- Usaremos **String** si la cadena de caracteres no va a cambiar.
- Usaremos **StringBuilder** si la cadena de caracteres puede cambiar y solamente tenemos un hilo de ejecución.
- Usaremos **StringBuffer** si la cadena de caracteres puede cambiar y tenemos varios hilos de ejecución.

En esta entrada utilizaremos StringBuilder teniendo en cuenta que todo lo que se explica aquí es aplicable a StringBuffer.

## Constructores de la Clase StringBuilder

Un objeto de tipo StringBuilder gestiona automáticamente su capacidad

– Se crea con una capacidad inicial.

– La capacidad se incrementa cuando es necesario.

La clase StringBuilder proporcionan varios **constructores**, algunos de ellos son:

| CONSTRUCTOR | DESCRIPCIÓN |
|---|---|
| StringBuilder () | Crea un StringBuilder vacío.<br>StringBuilder sb = new StringBuilder (); |
| StringBuilder(int n) | Crea un StringBuilder vacío con capacidad para n caracteres. |
| StringBuilder(String s); | Crea un StringBuilder y le asigna el contenido del String s.<br>String s = "ejemplo";<br>StringBuilder sb = new StringBuilder (s); |

## Métodos de la Clase StringBuilder

La clase StringBuilder proporcionan métodos para acceder y modificar la cadena de caracteres. Algunos de ellos son:

| MÉTODO | DESCRIPCIÓN |
|---|---|
| length() | Devuelve la longitud de la cadena |
| append(X); | Añade X al final de la cadena. X puede ser de cualquier tipo |
| insert(posicion, X) | Inserta X en la posición indicada. X puede ser de cualquier tipo. |
| setCharAt(posicion, c) | Cambia el carácter que se encuentra en la posición indicada, por el carácter c. |
| charAt(posicion) | Devuelve el carácter que se encuentra en la posición indicada. |
| indexOf('caracter') | Devuelve la posición de la primera aparición de carácter |
| lastIndexOf('caracter') | Devuelve la posición de la última aparición de carácter |
| substring(n1,n2) | Devuelve la subcadena (String) comprendida entre las posiciones n1 y n2 - 1. Si no se especifica n2, devuelve desde n1 hasta el final. |
| delete(inicio, fin) | Elimina los caracteres desde la posición inicio hasta fin. |
| reverse() | Invierte el contenido de la cadena |
| toString() | Devuelve el String equivalente. |

Los puedes consultar todos en la API de Java:

http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html
http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuffer.html

### Ejemplo de uso de la clase StringBuilder:

Vamos a escribir un método *separarMiles* que reciba un String que representa un número entero y devuelva un String con el mismo número al que se le añadirán los puntos separadores de millares.

Por ejemplo, si el método recibe el String "12345678" debe devolver el String "12.345.678"

Este problema lo podemos resolver de varias formas. En este caso la idea es darle la vuelta al número e insertar el primer punto en la cuarta posición del String, el siguiente punto 4 posiciones más adelante el siguiente otras 4 posiciones más adelante .... hasta llegar al final del número. De esta forma obtendremos grupos de 3 cifras separados por punto.

Finalmente le volvemos a dar la vuelta y ya lo tendremos.

Por ejemplo si el String es:

"12345678"

Primero le damos la vuelta:

"87654321"

Ahora tenemos que insertar un punto donde está el 5. Nos queda:

"876.54321"

Insertamos otro punto cuatro posiciones más adelante, donde está el 2:

"876.543.21"

Ahora intentaríamos insertar otro punto cuatro posiciones más adelante pero como llegamos al final el proceso termina.

Si le damos la vuelta obtendremos el resultado:

"12.345.678"

```
/*
 * Ejemplo de uso de StringBuilder
 * Separador de millares
 */
package string8;
public class String8 {

   public static void main(String[] args) {
      String s = "1234567890";
      s = separarMiles(s);
      System.out.println(s);
   }

   public static String separarMiles(String s){
      //creamos un StringBuilder a partir del String s
      StringBuilder aux = new StringBuilder(s);
      //le damos la vuelta
```

```
        aux.reverse();
        //variable que indica donde insertar el siguiente punto
        int posicion = 3;
        //mientras no lleguemos al final del número
        while(posicion < aux.length()){
            //insertamos un punto en la posición
            aux.insert(posicion,'.');
            //siguiente posición donde insertar
            posicion+=4;
        }
        //le damos de nuevo la vuelta
        aux.reverse();
        //el StringBuilder se pasa a String y se devuelve
        return aux.toString();
    }
}
```

## Eficiencia de la Clase StringBuilder frente a la Clase String

Podemos comprobar que es más eficiente utilizar StringBuilder frente a String realizando la siguiente prueba:

Vamos a concatenar un número grande de cadenas de caracteres, por ejemplo 100000, y vamos a medir el tiempo que se emplea en hacerlo.

Lo vamos a realizar primero utilizando la clase String. A continuación utilizando la clase StringBuilder y finalmente lo vamos a hacer utilizando StringBuilder pero asignando inicialmente memoria para la longitud final de la cadena resultante.

```
public class String3 {

    public static void main(String[] args) {
        String s = "cadena";
        long t1, t2;
        int n = 100000;

        System.out.print("Concatenar " + n + " cadenas con String: ");
        t1 = System.currentTimeMillis();
        concatenar(s,n);
```

```
        t2 = System.currentTimeMillis();
        System.out.println((t2-t1) + " milisegundos");

        System.out.print("Concatenar " + n + " cadenas con StringBuilder: ");
        t1 = System.currentTimeMillis();
        concatenar1(s,n);
        t2 = System.currentTimeMillis();
        System.out.println((t2-t1) + " milisegundos");

        System.out.print("Concatenar " + n + " cadenas con StringBuilder Optimizado: ");
        t1 = System.currentTimeMillis();
        concatenar2(s,n);
        t2 = System.currentTimeMillis();
        System.out.println((t2-t1) + " milisegundos");


    }

    //método que concatena n cadenas usando la clase String
    public static String concatenar(String s, int n) {
        String resultado = s;
        for (int i = 1; i < n; i++) {
            resultado = resultado + s;
        }
        return resultado;
    }

    //método que concatena n cadenas usando la clase StringBuilder
    public static String concatenar1(String s, int n) {
        StringBuilder resultado = new StringBuilder(s);
        for (int i = 1; i < n; i++) {
            resultado.append(s);
        }
        return resultado.toString();
    }

    //método optimizado que concatena n cadenas usando la clase StringBuilder
    //se crea un StringBuilder inicial con el tamaño total del String resultante
```

```
    public static String concatenar2(String s, int n) {
        StringBuilder resultado = new StringBuilder(s.length() * n);
        for (int i = 0; i < n; i++) {
            resultado.append(s);
        }
        return resultado.toString();
    }
}
```

La salida del programa dependerá del ordenador que utilicemos. En cualquier caso nos debe mostrar que concatenar utilizando String es más lento que si utilizamos StringBuilder. En mi caso el resultado obtenido ha sido este:

```
Concatenar 100000 cadenas con String: 304435 milisegundos
Concatenar 100000 cadenas con StringBuilder: 15 milisegundos
Concatenar 100000 cadenas con StringBuilder Optimizado: 1 milisegundos
```

# Clase StringTokenizer

La clase StringTokenizer sirve para separar una cadena de caracteres en una serie de elementos o **tokens.**

Se incluye en el paquete java.util

Los tokens se separan mediante delimitadores.

**Los delimitadores por defecto son:**

espacio en blanco
tabulador \t
salto de línea \n
retorno \r
avance de página \f

Un objeto StringTokenizer se construye a partir de un objeto String.

Para obetener los tokens del String podemos utilizar los métodos hasMoreTokens() y nextToken().

**hasMoreTokens()** devuelve true si hay más tokens que obtener en la cadena.

**nextToken()** devuelve un String con el siguiente token. Lanza una excepción del tipo NoSuchElementException si no hay más tokens.

Otro método importante es **countTokens()** que devuelve la cantidad de tokens que aun quedan por extraer

**Ejemplo** de uso de StringTokenizer:

```
String s = "blanco, rojo, verde y azul";
StringTokenizer st = new StringTokenizer(s);
while (st.hasMoreTokens())
        System.out.println(st.nextToken());
```

La salida que se obtiene por pantalla es:

blanco,

rojo,

verde

y

azul

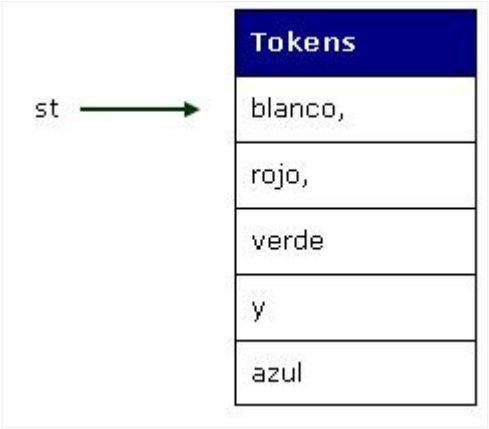
Se ha separado el String s en tokens separados por el delimitador por defecto. En este caso el espacio en blanco.

Si lo representamos de forma gráfica, el funcionamiento es el siguiente:

La instrucción

```
StringTokenizer st = new StringTokenizer(s);
```

Produce lo siguiente:




Se separa el String en tokens y el objeto st *apunta* al primero.

A continuación

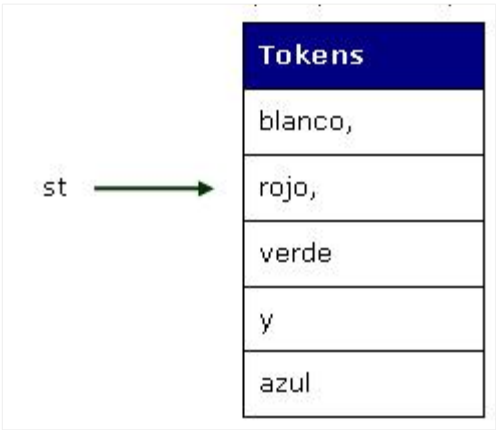
st.hasMoreTokens()

comprueba si hay tokens que extraer. En este caso st apunta a un token por lo tanto hasMoreTokens() devuelve true.

A continuación

st.nextToken()

Obtiene el token al que apunta st (blanco,) y avanza al siguiente.



El ciclo while repite el proceso.

Cuando se alcanza el último token y se avanza el siguiente la condición del while será false.

Los **delimitadores** se pueden especificar cuando se crea el objeto StringTokenizer.

Por ejemplo, para indicar que los delimitadores son la coma y el espacio en blanco:

StringTokenizer st = new StringTokenizer("colores rojo, verde y azul", ", ");

La ejecución del while anterior obtendría la salida:

colores

rojo

verde

y

azul

la coma no aparece ya que se ha especificado como delimitador y los delimitadores no aparecen.

Si queremos que aparezcan se debe escribir true como tercer argumento en el constructor:

StringTokenizer st = new StringTokenizer("colores rojo, verde y azul", ", ", true);

En este caso la salida es:

colores

rojo

,

verde

y

azul

# Clase Object Java. Método toString. Método equals.

La clase Object es la clase base del árbol de jerarquía de clases de Java.

**Todas las clases Java heredan directa o indirectamente de la clase Object.**

Esta clase define el comportamiento básico que todos los objetos deben tener. Para ello proporciona una serie de métodos que son heredados y por lo tanto están disponibles en todas las clases Java.

De todos los métodos que se heredan de Object en esta entrada vamos a ver los métodos toString y equals

## Método toString()

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

El método toString() devuelve un String que representa el objeto. El contenido de este String depende del objeto sobre el que se esté aplicando.

El método toString() **se invoca de forma automática cuando se nuestra un objeto mediante la instrucción System.out.println o System.out.print**

Esto quiere decir que las siguientes instrucciones son equivalentes:

```
System.out.println(unObjeto.toString());
System.out.println(unObjeto);
```

### Ejemplo de uso del método toString()

Disponemos de la siguiente clase llamada Coordenadas

```
package tostring1;
public class Coordenadas {
   private double x;
   private double y;

   public Coordenadas() {
   }
   public Coordenadas(double x, double y) {
      this.x = x;
      this.y = y;
   }
   public double getX() {
```

```
    return x;
  }
  public void setX(double x) {
    this.x = x;
  }
  public double getY() {
    return y;
  }
  public void setY(double y) {
    this.y = y;
  }
}
```

Un programa que utilice esta clase podría ser este:

```
package tostring1;
public class ToString1 {
  public static void main(String[] args) {
    Coordenadas punto1 = new Coordenadas(6.8,15.22);
    System.out.println(punto1);
  }
}
```

Se ha creado un objeto punto1 y se quiere mostrar por pantalla mediante la instrucción

System.out.println(punto1);

Como la clase Coordenadas no contiene un método toString(), la instrucción System.out.println(punto1); invocará al método toString() heredado de Object y mostrará el String que devuelve dicho método.

En este caso se mostrará por pantalla algo similar a esto:

```
tostring1.Coordenadas@4a5ab2
```

Vemos que el String que ha devuelto el método toString() heredado de Object contiene el nombre del package seguido del nombre de la Clase y de un número.

Como esto normalmente no es lo que queremos que aparezca cuando se muestra el contenido de un objeto, el métod toString() **lo podemos redefinir o sobrescribir (override)** para representar nuestros objetos de forma más adecuada.

Cuando se sobrescribe un método heredado hay que mantener el prototipo o encabezado del método tal cual se hereda. Si se escribe el método de forma distinta (por ejemplo, añadiendo un parámetro) estaremos sobrecargando el método, no sobrescribiéndolo.

En este caso lo vamos a modificar para que muestre el valor de las coordenadas de modo que cuando escribamos la instrucción:

System.out.println(punto1);

Se muestre por ejemplo:

Coordenadas: x = 6.8 y = 15.22

Para esto tenemos que escribir el método toString() dentro de la clase de la siguiente forma:

```
@Override
public String toString() {
        return "Coordenadas: " + "x = " + x + " y = " + y;
}
```

La anotación:

@Override

Es necesario escribirla antes del método e indica que estamos modificando el método toString() heredado de Object.

Después de añadir el método a la clase, la instrucción:

System.out.println(punto1);

Mostrará por pantalla:

Coordenadas: x = 6.8 y = 15.22

## Método equals()

```
public boolean equals( Object obj ){
   .......
}
```

Este método compara dos objetos para ver si son o no iguales.

Para determinar si son o no iguales, el método equals comprueba si los objetos pertenecen a la misma clase y contienen los mismos datos. Devuelve true si los objetos son iguales y false si no lo son.

Comparar dos objetos con equals **es distinto a hacerlo con el operador ==**.

Este operador == compara si dos referencias a objetos *apuntan* al mismo objeto.

Igual que ocurre con el método toString() en una clase podemos **sobrescribir el método equals()** para adaptarlo a nuestras necesidades.

Mediante la Opción *Insertar Código* de los entornos de desarrollo como NetBeans o Eclipse (Fuente -> Insertar Código en NetBeans) podemos añadir a nuestras clases el código de estos métodos heredados de Object.

El método equals que genera NetBeans de forma automática para la clase Corrdenadas es este:

```
@Override
public boolean equals(Object obj) {
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Coordenadas other = (Coordenadas) obj;
    if (Double.doubleToLongBits(this.x) != Double.doubleToLongBits(other.x)) {
        return false;
    }
    if (Double.doubleToLongBits(this.y) != Double.doubleToLongBits(other.y)) {
        return false;
    }
    return true;
}
```

En este caso este método equals nos sirve para realizar de forma corracta la comparación entre objetos de la clase Coordenadas y no hay que hacer ninguna modificación.

Vemos que lo primero que se comprueba es que se reciba un objeto para compararlo con el que llama al método, o sea, si el parámetro obj es igual a null significa que no apunta a ningún objeto y por lo tanto la comparación es falsa.

Lo siguiente es comprobar que ambos objetos son de la misma clase. **La clase a la que pertenece un objeto la podemos obtener mediante el método getClass()**. Si no lo son la comparación es falsa.

Por último nos queda comprobar los atributos de cada objeto para ver si contienen los mismos valores. Para ello el objeto que se recibe como parámetro se *convierte* a objeto de la clase Coordenadas y se comparan los atributos x e y de ambos.

Un ejemplo de uso del método equals podría ser este:

```
package tostring1;
```

```
public class ToString1 {
   public static void main(String[] args) {
      Coordenadas punto1 = new Coordenadas(6.8, 15.22);
      Coordenadas punto2 = new Coordenadas(6.8, 10.12);
      if(punto1.equals(punto2)){
         System.out.println("Coordenadas iguales");
      }else{
         System.out.println("Coordenadas distintas");
      }

   }
}
```

En este caso se mostrará el mensaje *Coordenadas distintas*ya que la coordenada *y* es diferente en ambos objetos y por tanto comparación dentro del método equals de la coordenada *y* dará como resultado que el método devuelva false.

# Clases Envolventes

Java es un lenguaje de programación orientado a objetos. Un programa Java debe contener objetos y las operaciones entre ellos. La única excepción a esto en un programa Java son los tipos de datos primitivos (int, double, char, etc.)
Los tipos de datos primitivos no son objetos pero en ocasiones es necesario tratarlos como tales. Por ejemplo, hay determinadas clases que manipulan objetos (ArrayList, HashMap, …). Para poder utilizar tipos primitivos con estas clases Java provee las llamadas clases envolventes también llamadas clases contenedoras o wrappers.
Cada tipo primitivo tiene su correspondiente clase envolvente:

| Tipo primitivo | Clase Envolvente |
|---|---|
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| char | Character |
| boolean | Boolean |

Estas clases proporcionan métodos que permiten manipular el tipo de dato primitivo como si fuese un objeto.

Las conversiones entre los tipos primitivos y sus clases envolventes son automáticas. No es necesario hacer un casting. Para realizarlas se utiliza el **Boxing/Unboxing**.

**Boxing**: Convertir un tipo primitivo en su clase Wrapper.

**Unboxing**: Convertir un objeto de una clase Wrapper en su tipo primitivo.

**Ejemplo de Boxing:**

```
double x = 29.95;

Double y;
```

```
y = x; // boxing
```

**Ejemplo de Unboxing:**

```
double x;

Double y = 29.95;

x = y; // unboxing
```

## Clase Integer

En la siguiente tabla aparecen algunos métodos de la clase Integer. El resto de clases envolventes correspondientes a tipos primitivos numéricos tienen métodos similares.

| | |
|---|---|
| Integer(int valor) | Constructor a partir de un int<br>Integer n = new Integer(20); |
| Integer(String valor) | Constructor a partir de un String<br>String s = "123456";<br>Integer a = new Integer(s); |
| int intValue()<br>float floatValue()<br>double doubleValue()<br>... | Devuelve el valor equivalente<br>Integer n = new Integer(30);<br>int x = n.intValue();<br>double y = n.doubleValue(); |
| int parseInt(String s) | Método estático que devuelve un int a partir de un String.<br>String s = "123456";<br>int z = Integer.parseInt(s); |
| String toBinaryString(int i)<br>String toOctalString(int i)<br>String toHexString(int i) | Métodos estáticos que devuelven un String con la representación binaria, octal o hexadecimal del número.<br>int numero = 12;<br>String binario = Integer.toBinaryString(numero); |
| Integer valueOf(String s) | Método Estático. Devuelve un Integer a partir de un String.<br>Integer m = Integer.valueOf("123"); |

API de la clase Integer: http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html

## Clase Character

Provee una serie de métodos para manipular los datos de tipo char. En la siguiente tabla aparecen algunos de estos métodos.

| | |
|---|---|
| Character(char c) | Constructor a partir de un char<br>char car = 'x';<br>Character a = new Character(car); |

| | |
|---|---|
| char charValue() | Devuelve el char equivalente<br>Character n = new Character('q');<br>char c = n.charValue(); |
| boolean isLowerCase(char ch) boolean isUpperCase(char ch)<br>boolean isDigit(char ch)<br>boolean isLetter(char ch) | Comprueba si es un carácter en minúsculas.<br>Comprueba si es un carácter en mayúsculas.<br>Comprueba si es un dígito (carácter del 0 al 9).<br>Comprueba si es una letra.<br>Todos son estáticos.<br>if(Character.isUpperCase(c)){<br>.....<br>} |
| char toLowerCase(char ch)<br>char toUpperCase(char ch) | Devuelve el char en mayúsculas.<br>Devuelve el char en minúsculas.<br>Métodos estáticos.<br>char car = 'u';<br>System.out.println(Character.toUpperCase(car));|
| Character valueOf(char c) | Método Estático. Devuelve un Character a partir de un char.<br>Character m = Character.valueOf('a'); |

API de la clase Character: http://docs.oracle.com/javase/7/docs/api/java/lang/Character.html

Ejemplo de uso de la clase Character: programa que lee un texto por teclado y muestra cuántos dígitos y letras contiene.

```
import java.util.Scanner;
public class JavaApplication325 {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      String texto;
      int cuentaCifras = 0, cuentaLetras = 0;
      System.out.println("Introduce texto ");
      texto = sc.nextLine();
      for (int i = 0; i < texto.length(); i++) {
         if (Character.isDigit(texto.charAt(i))) {
            cuentaCifras++;
         } else if (Character.isLetter(texto.charAt(i))) {
            cuentaLetras++;
         }
      }
      System.out.println("El texto contiene " + cuentaCifras + " dígitos");
      System.out.println("El texto contiene " + cuentaLetras + " letras");
   }
}
```

# Java Enum. Enumerados en Java

Un enum, también llamado enumeración o tipo enumerado es un tipo de dato definido por el usuario que solo puede tomar como valores los definidos en una lista.
Un enum se declara de forma general:

[modificadores] enum nombreEnum {VALOR1, VALOR2, VALOR3, ...}

**modificadores** (opcional) puede ser public, private, protected además de static.
**enum** es la palabra reservada para definir enumeraciones en Java.
**nombreEnum** es el nombre del nuevo tipo creado.
**VALOR1, VALOR2, ...** son los valores que pueden tomar las variables que se declaren de este tipo.
Se pueden declarar enumeraciones dentro o fuera de una clase pero **nunca dentro de un método**.
**Ejemplo:**
public enum Dia {LUNES, MARTES, MIÉRCOLES, JUEVES, VIERNES, SABADO, DOMINGO}
Por convenio se suelen escribir los valores en mayúsculas ya que se trata de constantes.
Una vez creado el tipo enum ya podemos declarar variables.
Para declarar una variable de tipo Dia:
Dia d;
La variable d solo podrá tomar uno de los valores definidos en la lista de valores.
Para darle un valor a las variables:
nombreDelEnum.VALOR;
Por ejemplo:
d = Dia.JUEVES;

**Ejemplo de uso**:

```
public class Enum1 {
   enum Dia {LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO}
   public static void main(String[] args) {
      Dia d; // declaramos una variable del tipo Dia
      d = Dia.DOMINGO; //asignamos un valor
      if (d == Dia.DOMINGO || d == Dia.SABADO)  //comparamos valores
         System.out.println("Estamos en fin de semana");
      else
         System.out.println("Aún no ha llegado el fin de semana");
      switch (d) {  //ejemplo de switch
         case LUNES:
         case MARTES:
         case MIERCOLES:
```

```
        case JUEVES:
        case VIERNES:
            System.out.println("Aún no ha llegado el fin de semana");
            break;
        default:
            System.out.println("Estamos en fin de semana");
    }
  }
}
```

Los tipos enumerados en Java son mucho más potentes que sus equivalentes en lenguajes como C++:

**Un enum en Java es una Clase.**

En general para utilizarlos los escribiremos en una clase con el mismo nombre del enum que añadiremos a nuestro proyecto

```
//Archivo Dia.java
public enum Dia{
        LUNES, MARTES, MIÉRCOLES, JUEVES, VIERNES, SABADO, DOMINGO
}
```

Aunque ya hemos visto antes que esto no es necesario y podemos escribirlos en el mismo archivo de la clase donde los vamos a utilizar.

Si declaramos el enum en el mismo archivo que la clase que lo usa y fuera de esta clase debemos tener en cuenta que en un archivo .java solo puede haber una clase pública y aquí también se incluyen las enumeraciones.

Por ser una clase, un enum en Java **puede contener atributos y métodos**.
Cada constante del enum es un objeto de la clase.

Todas las clases enum declaradas heredan de forma implícita de la clase Enum de Java (java.lang.Enum)

Aunque un enum es una clase, no se pueden crear instancias de un enum.

**Algunos métodos que se heredan de Enum:**

**name()**

*public final String name()*

Devuelve un String con el nombre de la constante que contiene tal y como aparece en la declaración.

### ordinal()

*public final int ordinal()*

Devuelve un entero con la posición de la constante según está declarada. A la primera constante le corresponde la posición cero.

### toString()

*public String toString()*

Devuelve un String con el nombre de la constante que contiene tal y como aparece en la declaración.
Sobrescribe el método toString de la clase Object.

### equals()

*public final boolean equals(Object other)*

Devuelve true si el valor de la variable enum es igual al objeto que recibe.
Sobrescribe el método equals de la clase Object.

### compareTo()

*public final int compareTo(Enum other)*

Compara el enum con el que recibe según el orden en el que están declaradas las constantes. Devuelve un número negativo, cero o un número positivo según el objeto sea menor, igual o mayor que el que recibe como parámetro.
Solo se pueden comparar enumeraciones del mismo tipo.

### valueOf()

*public static enumConstant valueOf(String s)*

Devuelve la constante que coincide exactamente con el String que recibe como parámetro.

### values()

*public static enumConstant [] values()*

Devuelve un array que contiene todas las constantes de la enumeración en el orden en que se han declarado. Se suele usar en bucles for each para recorrer el enum.
**Ejemplo** de uso de los métodos heredados de la clase Enum:

```
public class Enum2 {

   public enum Opcion {UNO, DOS, TRES, CUATRO}

   public static void main(String[] args) {
```

```
        Opcion op = Opcion.DOS;
        Opcion op2 = Opcion.CUATRO;
        if(op.name().equals("DOS"))
          System.out.println("Cadena DOS");
        System.out.println(op.ordinal());
        if(op2.compareTo(op)>0)
           System.out.println(op2 + " > " + op);
        String cadena = "UNO";
        if(Opcion.valueOf(cadena) == Opcion.UNO)
           System.out.println("Cadena UNO");
        for(Opcion x : Opcion.values())
           System.out.println(x);
    }
}
```

En Java podemos declarar constantes de un enum relacionadas con un valor.
Para ello la clase enum debe tener un atributo para cada valor relacionado con la constante y un constructor para asignar los valores.
Esos valores se pasan al constructor cuando se crea la constante.
El constructor debe tener acceso private o package. No puede ser público.
Las constantes se deben definir primero, antes que cualquier otro atributo o método.
Cuando un enum tiene atributos o métodos la lista de constantes debe acabar con punto y coma.
**Ejemplo** de constantes enum con valores asociados. Los valores 100 y 80 que acompañan a las constantes se pasan al constructor.

```
//Archivo Precios.java
public enum Precios {PRECIONORMAL(100), PRECIOVIP(80);
    double precio;
    Precios(double p){
       precio = p;
    }
   public double getPrecio() {
      return precio;
   }
}
```

```
//Clase Principal
public class Enum3 {
   public static void main(String[] args) {
```

```
        Precios p = Precios.PRECIOVIP;     //precio = 80
        System.out.println(p.getPrecio()); //muestra 80
    }
}
```