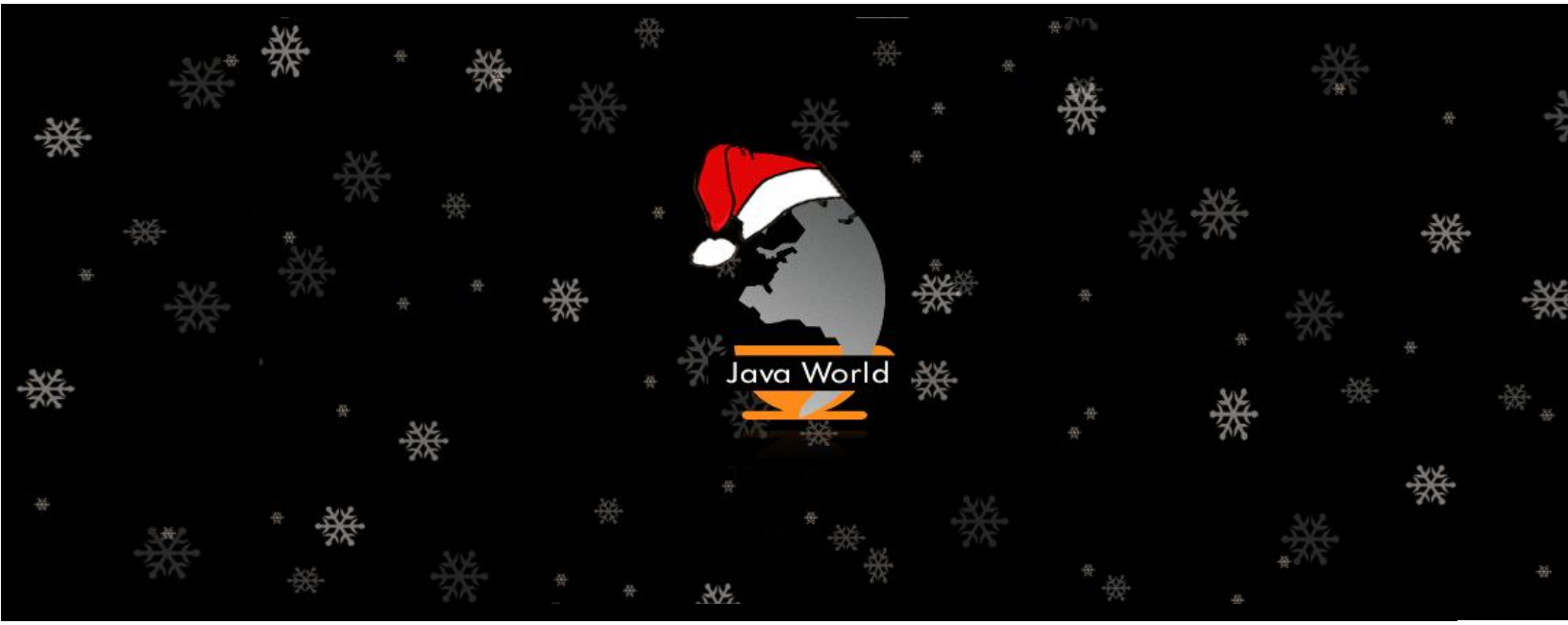
Capítulo 3

Literales – Inicialización - Wrappers

¿Querés ser un SCJP?

Revista Digital - Hecha por estudiantes para estudiantes



Novedades

Desde los creadores de JavaWorld les queremos desear unas felices fiestas a todos nuestros lectores y aprovechar esta oportunidad para dar la bienvenida al redactor de la nueva sección y diseñador de JavaWorld, al estudiante de Ingeniría Industrial Leonardo Blanco.

A partir de este número y cada semana nos estará presentando notas relacionadas con el mundo de la informática y el software, desde el punto de vista organizacional, economico e imagen corporativa.

En este capítulo veremos “como agregar valor a nuestros productos”.

> “En el mundo de lo intangible que es un software es necesario tener una buena imagen para justificar el precio”
>
> Leonardo Blanco

Stack y Heap

Para el examen solo es necesario que sepas lo siguiente:

- Variables locales se encuentran en el stack.
- Variables de instancia y objetos se encuentran en el heap.

Valores literales para todos los tipos primitivos

Un valor literal es simplemente una manera de representar el tipo de dato primitivo.

Integer

Hay tres maneras de representar un número entero en Java

- Decimal (base 10)
- Hexadecimal (base 16)
- Octal (base 8)

Literales decimal

Simplemente es escribir el número sin ningún prefijo.

Literales octal

Se antepone al número que deseamos escribir el `0`. Para formar el 9 (en decimal) se escribe `0`11 (en octal)

Literal hexadecimal

Los número hexadecimales son construidos utilizando valores numéricos y alfabéticos, siendo el orden el siguiente:
0 1 2 3 4 5 6 7 8 9 A B C D E F
NOTA: en los valores exadecimales, las letras pueden ir tanto en mayúsculas como en minúsculas.
Se antepone el `0x` para identificar un valor hexadecimal.

Punto flotante

- Se utiliza el `.` como separador de decimales.
- Por defecto un literal de punto flotante es interpretado como de tipo `double`.
- Para que sea interpretado como un `float` se debe agregar la `F` al final del literal (12.35`F`).
- Idem. con la `D` para que se interprete como un `double`, aunque no es necesario, dado que es el tipo que se le da por defecto.

Booleanos

Solo pueden ser `true` o `false`.

Caracteres

- Se escriben entre `‘’` (comillas simples).
- Un carácter es un `int` sin signo.
- Es legal asignarle un valor numérico a un caracter.
- Si el rango es mayor a un integer, o es negativo, no da error, pero es necesario castearlo para impedir la perdida de datos (loss of precision).
- Para especificar caracteres con un significado especial (salto de línea, tabulaciones, etc.) se utiliza la `\` seguido de un identificador (`u` para Unicode, `n` para salto de línea, `t` para tabulador).

Strings

Es una representación de un objeto String.

- Se escriben entre `“”` (comillas dobles).
- Representan un objeto, pero se pueden escribir como literales.

Operadores de asignación

Recordemos que una variable puede contener una de dos cosas:

- El valor del dato.
- La posición de memoria donde se encuentra el objeto (puntero).
    - Puede contener un valor **null**, que indica que no apunta a ninguna posición.

Asignación de valores primitivos

Se utiliza el operador **=**. Simplemente evalúa la expresión y/o literal que se encuentre del lado derecho, y asigna el resultado a la variable especificada en el lado izquierdo:

```
int a = 1;
int b = 2 * a;
int c = a / b;
```

Si se utiliza simplemente un literal del lado derecho del **=**, este puede ser casteado al valor de la variable, siempre que sea posible.

Toda expresión que involucre algún valor de tipo **int** o más pequeño, siempre será un **int**.

Casteo de primitivas

El casteo puede ser implícito o explícito.

- Un casteo implícito significa que no es necesario escribir ningún código, la conversión se efectúa automáticamente.
- Un casteo explícito, por el contrario, requiere de una codificación.

Las reglas son simples:

- Si la variable a asignar es de menor o igual tamaño a la variable donde será asignada, se produce un casteo implícito.
- Si la variable a asignar es de mayor tamaño a la variable donde será asignada, se debe realizar un casteo explícito, o de lo contrario, se generará un error de compilación.

Tabla de casteos implícitos para primitivas

La siguiente tabla indica la posición de variables primitivas, y cuáles de ellas puede contener. La interpretación es: una variable siempre puede contener todas las variables desde su posición (columna 1), hasta las inferiores, pero nunca superiores. Para ello es necesario un casteo explícito.

Posición	Tipo de dato	Puede contener
1	**Byte**	Posición actual + anteriores.
2	**Short**	Posición actual + anteriores.
3	**Char**	Posición actual
4	**Int**	Posición actual + anteriores.
5	**Long**	Posición actual + anteriores.
6	**Float**	Posición actual + anteriores.
7	**Double**	Posición actual + anteriores.

El tipo de dato **boolean** solo puede contener su mismo tipo.

Para darles un ejemplo, una variable de tipo **int** puede contener un **int**, **char**, **short** o **byte**.

Asignación de números con punto flotante

- Todo número de punto flotante es implícitamente de tipo double.

Asignación de un literal que es más grande que la variable

En el caso de que la primitiva sea más grande que el valor máximo permitido, existen dos alternativas:

- Si el resultado es casteado, se trunca la variable, y esta es almacenada con éxito.
- Si el resultado no es casteado, se genera un error de compilación "Error. Possible loss of precision".

Cuando se trunca una variable, simplemente se eliminan las posiciones extra a la izquierda, siendo por ejemplo:
int 128 equivale a 00000000 00000000 00000000 10000000, cuando casteamos a byte, nos queda 10000000 (el resto de la izquierda es eliminado).
Los operadores **+=**, **-=**, ***=**, **/=** realizan un casteo implícito.

Asignar una variable primitiva a otra variable primitiva

Cuando asignamos una variable primitiva a otra variable primitiva, simplemente se copia el valor de esta, quedando ambas como equivalentes, pero totalmente independientes. Esto quiere decir que si modificamos una, la otra no se verá afectada.

Ámbito de las variables (scope)

Básicamente, el ámbito indica por cuánto tiempo la variable existirá. Existen cuatro tipos diferentes de ámbitos:

- Variables estáticas — Son creadas cuando la clase es cargada, y existirán mientras que la clase siga cargada en la JVM.
- Variables de instancia — Son creadas cada vez que una nueva instancia es inicializada, y existirán mientras que la instancia exista.
- Variables locales — Estas existirán durante el tiempo que el método permanece en el stack.
- Variables de bloque — Estas existirán mientras que se esté ejecutando el bloque de código.

```
public class AlgunaClase {

  private static int valorEstatico;
  public static void getValorEstatico() {
    return valorEstatico;
  }

  private int valorInstancia;

  public AlgunaClase() {
    valorInstancia = 0;
  }

  public void setValorInstancia(int valor) {
    valorInstancia = valor;
  }

  public int getValorInstancia() {
    return valorInstancia;
  }

  public void otroMetodo(int valor) {
    valor++;
    for (int x = 0 ; x < 100 ; <++ ) {
      //Código...
    }
  }
}
```

Variables estáticas

Variables de instancia

Variables locales

Variables de bloque

Errores más comunes con problemas de scope

- Intentar acceder a un variable de instancia desde un contexto estático.
- Intentar acceder a una variable local desde un método anidado (un método que llama a otro, y el segundo utiliza una variable local del primero).
- Intentar utilizar una variable de bloque, luego de haber concluido la ejecución del bloque.

Utilizar una variable o array que no ha sido inicializado ni asignado

Java permite inicializar una variable declarada o dejarla sin inicializar. Cuando intentemos utilizar dicha variable (en el caso de no haberla inicializado), podemos encontrarnos con diferentes comportamientos dependiendo del tipo de variable o array con que estemos trabajando, así como, el ámbito de la variable.

Variables de instancia de tipos primitivos y objetos

Las variables de instancia son inicializadas a un valor por defecto. La siguiente tabla muestra los valores de inicialización según el tipo:

Tipo de variable	Valor por defecto
Referencia a objeto	null (no referencia a ningún objeto)
`byte`, `short`, `int`, `long`	0
`float`, `double`	0.0
`boolean`	false
`char`	'\u0000'

Variables de instancia de tipo array

Cuando se declara una variable de este tipo, pueden darse dos casos de no inicialización:

- Si no se inicializa el array, el mismo contendrá una referencia a null (al fin y al cabo, un array es un objeto).
- Si se inicializa el array, pero a sus elementos no se le asignan valores (`int[] a = new int[10]`), los elementos del array serán inicializados al valor por defecto del tipo del array (en el caso del ejemplo 0).

Variables primitivas locales y objetos

Las variables locales son aquellas que hayan sido declaradas dentro de un método, y esto incluye a los parámetros del mismo. En este caso, el término automático indica variable local, no que se inicializa automáticamente.

Variables primitivas locales

Estas variables deberán de ser inicializadas siempre antes de utilizarlas, de lo contrario, obtendremos un error en tiempo de compilación "Variable X may not have been initialized".
Otro problema que puede llegar a surgir es el de inicializar la variable dentro de un bucle condicional. El compilador detecta esto, e indica con un error de compilación el posible problema "Variable X might not have been initialized".

Variables de referencia locales

Si se declara una variable de referencia local, pero no se inicializa, por más que luego solo se haga una verificación de si vale `null`, esto arroja un error. Esto se debe a que `null`, también es un valor.

Variables de array locales

Se comportan igual que las referencia a objetos locales, con la salvedad de que si una variable array a sido inicializada, y se le asigna un conjunto de elementos de tipo X, pero no los valores, estos siempre serán inicializados al valor por defecto del tipo, sin importar el ámbito de la variable.

Envío de variables a métodos

Los métodos pueden ser declarados con parámetros de tipo primitivos y/o referencias a objetos. Es importante conocer el efecto de modificar cualquiera de los dos tipos de parámetro, y los efectos de realizar esto.

Envío de variables de referencia

Cuando se pasa como argumento una referencia a un método, lo que se está enviando es la dirección de memoria a la que apunta la referencia, y no el objeto en sí. Esto quiere decir que ambas variables estarán referenciando al mismo objeto en el heap, de manera que si el objeto es modificado dentro del método, se verá afectado fuera de este también.

Java utiliza la semántica de paso por valor?

Si. A pesar de que pareciera que no, dado lo anterior, en realidad se está pasando el valor contenido por la variable. Ahora, que este valor sea un tipo primitivo o una referencia no importa, al pasar un argumento, el parámetro se carga con una copia bit a bit.
Otra consideración es que el parámetro no puede modificar el valor del argumento, lo que es diferente de modificar el estado del objeto al que apuntan ambos.

Declaración, construcción e inicialización de arrays

En Java, un array es un objeto que almacena múltiples variables del mismo tipo. Estos pueden contener tanto tipos primitivos como referencias a objetos, pero el array en si mismo siempre será un objeto en el heap.
Para utilizar los arrays, es necesario saber tres cosas:

- Como crear una variable que referencie a un array (declaración).
- Como crear un objeto de tipo array (construcción).
- Como completar el array con elementos (inicialización).

Declaración de un array

Los array son declarados especificando el tipo de dato que contendrá el array seguido de `[]`. También se pueden especificar varias dimensiones de arrays, siendo cada par de `[]` equivalentes a una dimensión.

```
int[] array_1;          //Un array con primitivos
int[][] array_1;        //Un array con primitivos de 2 dimensiones (también conocido
                        //como matriz)
int[][][] array_1;      //Un array con primitivos de 3 dimensiones (también conocido
                        //como hipermatriz)
```

Construcción de un array

Construir un array significa crear el objeto en el heap. Para reservar el espacio en el heap, Java debe saber el tamaño del array, de manera que este debe ser especificado al momento de la creación.

Construir un array de una dimensión

Para construir (crear o inicializar son sinónimos válidos para esta acción) un array, se puede utiliza cualquiera de las siguientes sintaxis:

```
int[] array_1    = new int[5];      //Declara una variable de tipo referencia que
                                    //contiene un array de ints, e inicializa el array
                                    //para poder contener 5 variables de tipo int (son
                                    //inicializadas al valor por defecto).
Object[] array_2 = new Object[5];   //Idem. Tener en cuenta que los elementos del array
                                    //contienen una referencia a null (valor por
                                    //defecto). En ningún momento se crea un objeto de
                                    //tipo Object (salvo el Array en si mismo).
Object[] array_2;                   //Un array con referencias
```

Es necesario si utilizamos la sintaxis anterior especificar la cantidad de elementos que el array contendrá (1 es el mínimo).
Existe otra manera de construir un array, la cual es una forma abreviada para realizar esto último, y a la vez inicializarlo (esta se explica más adelante).

Construir un array multidimensional

Un array multidimensional no es más que un array de arrays.

```
int[][] array_1 = new int[5][]; //Es válido en Java, dado que solo se requiere saber
                                //el espacio necesario para la variable array_1
```

Inicialización de un array

Inicializar un array significa cargar los elementos del mismo.

```
int[] array_1 = new int[3];

int[0] = 23;    //Al elemento 1 se le da el valor 23
int[1] = 12;    //Al elemento 2 se le da el valor 12
int[2] = 45;    //Al elemento 3 se le da el valor 45
int[3] = 1;     //Ops. Si realizamos esto nos encontraremos con un error en tiempo de
                //ejecución "ArrayOutOfBoundsException".
                //El límite del array es [0;2]
```

Recordemos que **los límites de un array están comprendidos siempre entre 0 y total de elementos – 1**.

Inicializando elementos en un bucle

Los arrays tienen una única variable de instancia pública, la cual es el length. Esta variable indica cuantos elementos puede contener el array, pero no cuántos de ellos se encuentran inicializados.

```
class Persona {}

public class Program {
  static public void main(String[] args) {
    Persona[] arrayPersona = new Persona[5];

    for ( int x = 0 ; x < arrayPersona.length ; x++ ) {
    // x llegará a valer 4. Cuando su valor sea 5, la condición será 5 < 5 por lo que
    // no entra en el bucle y no se excede el límite del array
      Persona[x] = new Persona();
    // Inicializa cada elemento del array con un nuevo objeto persona
    }
  }
}
```

Declaración, construcción, e inicialización en una línea

Existen dos maneras de realizar estos pasos con un array:

```
//Forma 1 - Todo en una línea
int x = 12;
int[] array_1 = { 8, x, 22 };

//Forma 2 - Multiples líneas
int x = 12;
int[] array_1 = new int[3];
array_1[0] = 8;
array_1[1] = x;
array_1[2] = 22;
```

Ambas sintaxis son totalmente equivalentes.

Construyendo e inicializando un array anónimo

Permite construir un array e inicializarlo, sin asignarlo a ninguna variable. Pero, para que se podría utilizar? Es muy útil para enviar como argumento un array que se genere en el momento. La sintaxis es:

```
new int[] { 1, 14, 2, 4 };

int[] array_1 = new int[] { 1, 3 }; //A la variable array_1 se le pasa la referencia
                                    //del array anónimo.
```

Cuando se genera un array anónimo no debe especificarse la cantidad de elementos que contendrá. Estos son obtenidos de la cantidad de elementos inicializados entre {}.

Asignaciones legales de elementos de un array

Un array solo puede ser declarado de un tipo en particular, pero eso no quiere decir que todos los elementos que este contenga tengan que ser explícitamente del mismo tipo.

Arrays de tipos primitivos y referencia

Aplican las mismas propiedades para los elementos de un array que para una variable del mismo tipo.

Asignaciones legales de referencias de tipo array

Cuando hablamos de la referencia es al objeto array, y no a sus elementos.

Referencias para tipo de datos primitivos

Dado que un array es un objeto, los tipos de array para primitivos son objetos totalmente diferentes, que no comparten un árbol de herencia, por lo que no es legal asignar ningún tipo de array, siempre que no sea el declarado (no aplican los tipos válidos para conversiones implícitas).

Referencias para tipos de datos de referencia

En estos casos, se puede asignar lo siguiente:

- La referencia de array asignada puede ser de la misma clase o subclase (downcasting en el árbol de herencia).
- La referencia de array asignada no puede ser de una superclase (upcasting en el árbol de herencia).
- La referencia de array asignada puede ser una interfaz, pudiendo ser el tipo de la referencia cualquier clase que implemente la interfaz, o subclase que derive de una clase que implemento esta.

Referencias para tipos de datos de referencia en arrays multidimensionales

En estos casos, es necesario que el tipo de dato a asignar sea una referencia a un array de las mismas dimensiones que el tipo de dato.

```
int[] array = new int[4];
int[][] array2D = new int [5][];

array2D = array; //Error. Las dimensiones no concuerdan

int[] arrayOther = new int[15];
array = arrayOther; //OK. Las dimensiones y el tipo concuerdan.
```

Bloques de inicialización

Estos bloques permiten ejecutar lógica, fuera de un constructor o un método.
Existen dos tipos de bloques de inicialización: estáticos y de instancia.
Las reglas que aplican a estos son:

- Los bloques de inicialización se ejecutan en el orden en que aparecen (arriba hacia abajo).
- Los bloques de inicialización estáticos
  - Se ejecutan una sola vez, cuando la clase es cargada.
- Los bloques de inicialización de instancia
  - Se ejecutan cada vez que se crea una nueva instancia de la clase.
  - Se ejecutan luego de la llamada a `super()`.

Veamos un ejemplo de inicialización:

```
class Padre {
  static {
    System.out.println("Se ejecuta el bloque de instancia de la clase padre");
  }
  {
    System.out.println("Se ejecuta el bloque de inicialización de la clase padre");
  }
  public Padre() {
    System.out.println("Se ejecuta el contructor de instancia de la clase padre");
  }
}
class Hija extends Padre {
  static {
    System.out.println("Se ejecuta el bloque de instancia de la clase hija");
  }
  {
    System.out.println("Se ejecuta el bloque de inicialización de la clase hija");
  }
  public Hija() {
    System.out.println("Se ejecuta el contructor de instancia de la clase hija");
  }
}
public class Inicializacion {
  static public void main(String[] args) {
    new Hija();
  }
}
```

```
Se ejecuta el bloque de instancia de la clase padre
Se ejecuta el bloque de instancia de la clase hija
Se ejecuta el bloque de inicialización de la clase padre
Se ejecuta el contructor de instancia de la clase padre
Se ejecuta el bloque de inicialización de la clase hija
Se ejecuta el contructor de instancia de la clase hija
```

Las clases Wrapper

Estas clases sirven para dos propósitos funadmenales en Java:

- Otorgar un mecanismo para "envolver" (wrap) valores primitivos en un objeto.
- Otorgar una serie de funciones útiles para los valores primitivos.

Tabla con los diferentes wrappers:

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

Creación de objetos Wrapper

Es necesario comprender para el examen las tres maneras más comunes de crear un objeto Wrapper.
Algunas cuestiones a tener en cuenta con los Wrappers:

- Los objetos Wrapper son inmutables (no pueden modificar su valor una vez creados).
- Si se inicializa un Wrapper con un String para obtener un tipo primitivo, puede generar una Excepción de tipo NumberFormatException, si el String no corresponde con el tipo de dato esperado por el Wrapper.

Los constructores del Wrapper

La mayoría de los Wrappers proporcionan dos constructores, argumento del tipo primitivo, o String.

```
Integer i1 = new Integer(23);
Integer i1 = new Integer("23");

Float f1 = new Float(1.23F);
Float f1 = new Float("1.23F");
```

El método valueOf()

Este método estático otorga otra manera de crear un objeto Wrapper de un primitivo. En casi todos los Wrappers se encuentra sobrecargado.

- valueOf(`String` valor)
- valueOf(`String` valor, `int` base)

Utilidades de conversión de los Wrappers

Los métodos detallados a continuación son los más utilizados y los que se incluyen en el examen.

xxxValue()

Utilizado para convertir el valor de un Wrapper numérico en su tipo primitivo.

```
Integer i1 = new Integer(23);
int i = i1.integerValue();
```

Todos los Wrapper de los valores numéricos primitivos tienen los métodos xxxValue() para todos los primitivos numéricos.

parseXxx() y valueOf()

- Ambos toman como primer argumento un String para crear el valor
- Ambos arrojan una excepción "NumberFormatException"
- parseXxx()
  - Devuelve el valor primitivo de un Wrapper
  - Para cada Wrapper de tipo primitivo numérico, el método se obtiene reemplazando Xxx por el nombre del tipo primitivo con la primer letra en mayúscula.
- valueOf()
  - Devuelve un nuevo Wrapper del mismo tipo que el del método invocado
  - Existen en todos los Wrappers con tipo primitivo numérico

toString()

Este método se encuentra en la clase **Object**. Se utiliza para poder representar de manera más explícita un objeto.
Todos los Wrappers tienen este método sobre escrito y marcado como **final**. Devolviendo un **String** con el valor del primitivo que contienen.
A su vez, todos los Wrappers numéricos tienen un método estático sobrecargado, con un parámetro del tipo primitivo del Wrapper, devolviendo un objeto **String**.
También, **Integer** y **Long** tienen otro método estático sobrecargado que tiene dos parámetros, el primero del tipo primitivo del Wrapper, y el segundo de tipo **int**, que indica la base sobre la cual formatear el valor.

toXxxString() (Binario, Hexadecimal, Octal)

Para los Wrappers **Integer** y **Long** existen tres métodos:

- toOctalString()
- toHexString()
- toBinaryString()

Todos estos tienen un solo parámetro, que es el tipo primitivo del Wrapper.

Tabla de métodos de conversión del Wrapper

Método	s=static n= NFE	Boolean	Byte	Short	Character	Integer	Long	Float	Double
byteValue			X	X		X	X	X	X
shortValue			X	X		X	X	X	X
intValue			X	X		X	X	X	X
longValue			X	X		X	X	X	X
floatValue			X	X		X	X	X	X
doubleValue			X	X		X	X	X	X
parseXxx	s,n	X	X	X		X	X	X	X

parseXxx (con base)	s,n		X	X		X	X		
valueOf	s,n	X	X	X		X	X	X	X
valueOf (con base)	s,n		X	X		X	X		
toString		X	X	X	X	X	X	X	X
toString (primitivo)	s	X	X	X	X	X	X	X	X
toString (primitivo, base)	s					X	X		

Firmas de los métodos:

primitive xxxValue()	convierte un Wrapper en un primitivo
primitive parseXxx(String)	convierte un String en un primitivo
Wrapper valueOf(String)	convierte un String en un Wrapper

Autoboxing

Este término viene relacionado con unboxing, boxing, y sus respectivos automáticos. Estos son utilizados en el momento en que se utilizan Wrappers y primitivas. Para explicar mejor el concepto, mostramos un ejemplo:

```
//Este ejemplo realiza un unboxing y boxing automático
Integer wrapper = new Integer(78);
int primitive = wrapper.intValue();  //Unboxing - obtenemos la variable primitiva
primitive++;                         //Realizamos las operaciones necesarias sobre la
                                     //primitiva
wrapper = new Integer(primitive);    //Boxing - volvemos a contener a la primitiva en
                                     //el Wrapper

//La sintaxis anterior en Java 5 es equivalente a:
Integer wrapper = new Integer(78);
wrapper++;                           //Esto hace el unboxing, realiza la operación, y
                                     //el boxing
```

Boxing, ==, y equals()

- El operador !=
  - Este no se encuentra sobrecargado. Compara posiciones de memoria.
- El operador ==
  - Se encuentra sobrecargado. Compara valores
  - Cuando se compara un wrapper y una primitiva, al wrapper se le hace un outboxing.
- El método equals()
  - Compara valores.

La regla general para Wrappers y auto-unboxing y auto-boxing, es que, estos pueden ser utilizados como primitivas.

Un wrapper con una referencia null genereará un error cuando se intente realizar alguna operación (ya sea invocar a algún método, o alguna operación que genere un auto-unboxing).

Sobrecarga (overloading)

En esta sección agregaremos algunos conceptos a la sobrecarga:

- Widening
- Autoboxing
- Var-args

Cuando una clase tiene métodos sobrecargados, uno de los trabajos del compilador es determinar que método utilizar si encuentra una invocación a un método sobrecargado.
En cada caso, cuando no se encuentra una concordancia, la JVM utiliza el método con un parámetro más abarcativo que el de la llamada con la menor cantidad de parámetros.

Sobrecarga con boxing y var-args

El compilador prefiere utilizar variables más abarcativas, que realizar un unboxing.
Ejemplo:

```
public class UnboxingVsWidering {
    static public void figth(Integer i) { System.out.println("Unboxing wins!"); }
    static public void figth(long l) { System.out.println("Widering wins!"); }

    static public void main(String[] args) {
        int x = 45;
        UnboxingVsWidering.figth(x);
    }
}
```

La regla es:

- Widering le gana a boxing
- Boxing le gana a var-args

No es posible combinar widering and boxing. Si el compilador debiera realizar ambas acciones para encontrar una concordancia de método, se produce un error.

Sobrecarga con var-args

Al igual que widering y boxing, estos aplican de la misma manera para var-args.

Recolector de basura (Garbage collector)

El recolector de basura provee una solución automática a la administración de memoria.
La ventaja es que desliga al usuario de tener que crear la lógica para la administración de los recursos de memoria.
La desventaja es que no es posible controlar cuando se ejecuta y cuando no.

Vistazo del GC

El heap es el único lugar donde viven los objeto, y es el único lugar donde el GC hace efecto.
Para el examen es necesario saber que el GC se encarga de eliminar los objetos que no pueden ser accedidos por el programa en ejecución, y eliminarlos.

Cuando se ejecuta el GC?

El GC se encuentra bajo el control de la JVM. Esta última decide cuando ejecutar el GC. Es posible desde nuestro progama indicarle a la JVM que ejecute el GC, pero no hay garantía de que esto se realice.
Generalmente, la JVM ejecuta el GC cuando detecta que la memoria se encuentra baja.

Cómo funciona el GC?

No es ciencia cierta la manera en que funciona. Lo que si sabemos, es cuando un objeto pasa a ser candidato para el GC.
Un objeto es candidato para el GC cuando ningún ***thread*** *en ejecución puede acceder al objeto.*

Establecer un objeto para candidato al GC

Son algunas maneras para indicar al GC que el objeto debe ser marcado para su eliminación.

Referencia a null

Estableciendo la referencia del objeto a **null**, perdemos la posición de memoria, de manera que no es más accesible.

Reasignando la referencia

Cuando creamos un nuevo objeto, y el resultado de la creación es asignado a una referencia, el objeto que referenciaba anteriormente queda marcado como candidato para el GC.

Aislando una referencia

Cuando una clase tiene una variable de instancia del mismo tipo que la clase, y un objeto tiene cargado en dicha variable una instacia de si misma, a esto se le llama la “isla de la aislación”.
Ejemplo:

```
public class Isolation {
  private Isolation iso;

  public Isolation () {
    iso = this;
  }

  public static void main(String[] args) {
    Isolation is = new Isolation();
    is = null; //Aun existe una referencia al objeto creado, pero esta se encuentra
               //dentro de un objeto que no puede ser accedido.
  }
}
```

Lo que se viene

En la próxima entrega estaremos adentrándonos en el mundo de los operadores.

Veremos los diferentes tipos de operadores que esxisten en el mundo de Java y como utilizarlos.

Estudiaremos el operador instanceOf para saber si un objeto corresponde a la relación Is-A con una clase.

Aprenderemos sobre los operadores aritméticos, y daremos un vistazo a los operadores lógicos.

Forzar el GC

Esiste la posibilidad de solicitar a la JVM que ejecute el GC, pero esto es solo una sugerencia. La JVM puede o no realizar la acción. Esto se logra con `System.gc()`.

El método finalize()

Este método es ejecutado cuando un objeto está a punto de ser eliminado por el GC (no cuando se vuelve candidato para el GC).
Además, dado que no sabemos cuándo se ejecutará el GC, tampoco sabemos cuándo se invocará a dicho método.
Otras consideraciones sobre finalize:

Este solo podrá ser llamado una sola vez (lo invoca el GC).
La llamada a este método puede provocar que el objeto no sea eliminado.

Valor Creativo

Diseño de Informes Profesionales

La importancia de la Imagen

Por Leonardo Blanco

El público que probablemente lea esta revista seguramente sean personas que buscan diferenciarse de los demás programadores informáticos a partir de la base del conocimiento. Ahora:

¿Es el conocimiento la única cualidad que buscan las empresas o los clientes a la hora de elegir una persona o producto?
La respuesta es no. En la realidad importa más la portada que el contenido del libro y por ello hay que diferenciarse. La imagen en una empresa, producto o persona es el factor más valioso, y es necesario cuidarlo.

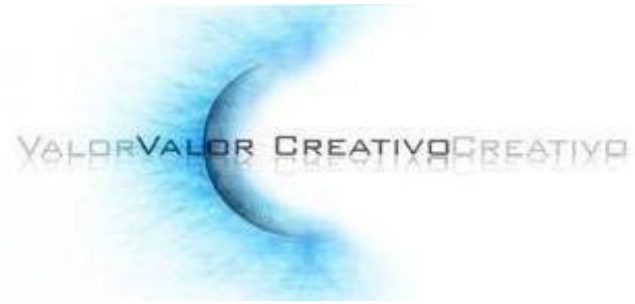
¿Por dónde se debe empezar a cuidar la imagen?
La respuesta la voy a dividir en dos instancias: estudiantes y profesionales. Los estudiantes deben enfocar su atención en el formato de sus informes y los profesionales deben incluir otros factores (apariencia, dialogo, etc).

¿Por qué hay que darle importancia a los formatos de un informe?

1. Estudiantes:

- Se demuestra que uno le dedico tiempo y dedicación al trabajo.
- Se le agrega una firma personal por sobre los demás.
- Te da cierto prestigio y una buena imagen (tratar que el formato sea tan bueno como el contenido).
- La primera impresión es la más importante, y esta entra a partir de los ojos.
- Un gran creativo dijo una vez el contenido es el 20% y el diseño es el 80% de importancia.
- Tarde o temprano se van a encontrar con un profesor que no lee el informe sino que los mira por arriba.
- Por último, hay profesores que no les importa el formato, pero, escuchen este consejo y no se lo olviden nunca:

¡Dedicarle tiempo al diseño de un Formato nunca está de más!



2. Profesionales:

- Cuando hay dos productos a mismo precio y de marcas desconocidas, el 67% de las personas escogen el producto de mejor diseño en estética y solo un 16% lee cuales son los beneficios del producto. En otras palabras usted puede perder 7 de cada 10 ventas por no haber invertido en el diseño.
- Cuando uno presenta un informe sobre un proyecto, no es lo mismo un informe que represente **la imagen de la empresa** que un papel con párrafos escritos y monótonos. La gente tiende a recordar más las imágenes que las palabras, esto ocurre porque el cerebro se divide en dos hemisferios, en donde uno retiene las primera y el otro las últimas. Por ello es conveniente convencer al cliente desde ambos frentes.
- La imagen de un profesional es el **valor agregado** en su venta de bienes y servicios, y se basa en los siguientes conceptos:
    - Buena apariencia.
    - Buen diálogo (tratar a las persona de usted y con su respectivo tecnicismo).
    - Portar siempre consigo mismo una **tarjeta personal**.
    - Que el diseño de los informes represente todo lo anterior.

VALOR CREATIVO

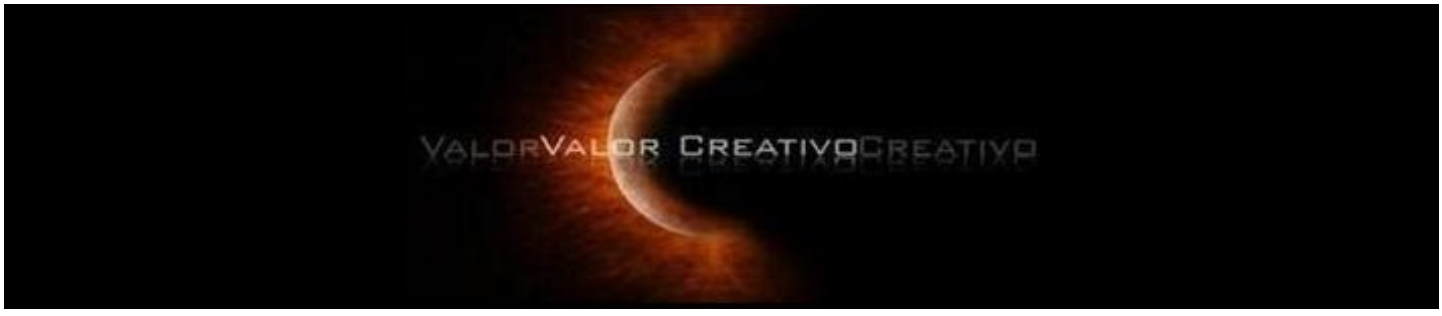
Diseño de Informes Profesionales

¿Que consejo se puede dar?

Para ser un profesional en algo la primer regla es saber algo que el otro desea saber. Ese conocimiento es el producto a vender y el entorno que proponemos es el envase.

Todos saben que es muy dificil poner precio a lo que no se puede tocar, en la Industria de la Informática ese intangible es el software. En el contexto actual el saber la respuesta a esta pregunta es tener la clave del éxito, por eso recomiendo hacer las cosas con la mejor calidad, adaptarse al entorno y a sus clientes, y por sobre todas las cosas, realizar sus trabajos y su entorno agradables a la vista.

Gracias por la oportunidad y los espero en ValorCreativo (http://valorcreativo.blogspor.com).



¿Tenés lo que se necesita para ser un SCJP?

Autores
Gustavo Alberola
Matias Álvarez

Diseño
Leonardo Blanco