Certificación JAVA

# Capítulo 1: Bloques de construcción Java (Java Building Blocks)

## Estructuras de las clases Java

* Para utilizar la mayoría de clases se tienen que crear objetos.
* Un objeto es una instancia en tiempo de ejecución de una clase en memoria.
* Una clase contiene variables (fields) y métodos.
* Hay 3 tipos de comentarios, línea simple (//), multilinea (/\* \*/), Javadoc (/\*\* \*\*/).
* Las clases no necesariamente tienen que ser públicas, pero si se declara pública el archivo tiene que llamarse igual que el nombre de clase.
* Si hay dos clases en el mismo archivo máximo una de ellas puede ser pública.
* Al compilar un archivo .java lo que se obtiene es un bytecode (.class).
* El método main siempre debe ser estático sino se tendrá un error, así como si al correr una clase sin main también lanzara una excepción.
* Los argumentos del main puede definirse: String[] args, o String args[], o String… args.
* Los parámetros enviados al main, al ejecutar una aplicación se separan con espacios, si deseas usar espacios en el parámetro enviado debes rodearlo con comillas.
* Los argumentos de una línea de comandos siempre son Strings aunque pongas un número.
* Si utilizas un argumento en el método que no haz enviado (por ejemplo args[1]) se lanzará una excepción por el array.
* Para compilar se necesita JDK, para solo ejecutar se necesita solo el JRE.

## Declaración de Paquetes e Imports

* Los paquetes podrían hasta ser a.b.c, las reglas son las mismas que la de los nombres de variables.
* Al colocar el .\* se obtienen todas las clases del paquete (no las clases dentro de paquetes hijos ni variables ni métodos), hay algo llamado **static import** pero se verá en el capítulo 4.
* Existe redundancia de imports, por ejemplo java.lang se importa automáticamente en la clase, importar cualquier clase dentro de este paquete o sus subpaquetes sería redundante. Tambien sería redundante importar clases del mismo paquete.
* Aquí tenemos algunos ejemplos de imports que no funcionan:

**import** java.nio.file.\*; // BIEN

**import** java.nio.file.Files; // BIEN

**import** java.nio.file.Paths; // BIEN

**import** java.nio.\*; // MAL – un comodín(wildcard \*) solo obtiene clases,

//no "file.\*Files"

**import** java.nio.\*.\*; // MAL – solo puedes tener un comodín(wildcard \*)

//y debe de estar al final

**import** java.nio.files.Paths.\*; // MAL – No puedes importer métodos

//solo clases

* También se tienen casos de conflictos de nombres, no pueden importarse clases con el mismo nombre ya sea explícito o con el comodín.

**import** java.util.\*;

**import** java.sql.\*; // NO COMPILA

**import** java.util.Date; //SI COMPILA

**import** java.sql.\*;

**import** java.util.Date;

**import** java.sql.Date; //NO COMPILA

**public** **class** Conflicts {

Date date;

// some more code

}

* Si se necesitase utilizar ambas clases, se deja una en los imports y la otra se utiliza con el nombre completo (paquete.clase) o utilizar ambos con el nombre completo.
* Al ejecutar un programa por consola no se coloca la extensión (.class) (java paquete.ClassA).
* Si la clase a ejecutar utiliza clases guardadas en otros directorios se importa las rutas al classpath:

java -cp ".:/tmp/someOtherLocation:/tmp/myJar.jar" myPackage.MyClass

* El punto al inicio indica que quieres incluir el directorio actual donde estas realizando la ejecución en el classpath, lo demás indica que se deben buscar clases y paquetes en someOtherLocation y dentro de myJar.jar.
* También se puede utilizar el comodín (wildcard \*) para agregar todos los jars dentro de un directorio, pero no agregará los jars dentro de un subdirectorio del directorio indicado (no subpaquetes de directoryWitgJars).

java -cp "C:\temp\directoryWithJars\\*" myPackage.MyClass

* Cuando en el examen, en fragmentos de código no se incluye imports debe revisarse si se tiene un número de línea explícito y que no comience en 1, porque se comienza en 1 y no tiene los imports que debería tener no compilará el código.
* Java toma como parámetro al nombre de la clase (sin el .class) al ejecutar.

## Creando Objetos

* Los constructores tienen las dos siguientes características básicas: mismo nombre de la clase y no tiene tipo de retorno.
* Todo lo que se encuentre dentro de llaves ( {} ) es llamado bloque de código (code block). Bloques de código fuera de métodos son llamados inicializadores de instancia (instance initializers) (Capítulo 5).
* Hasta el momento el orden de ejecución de campos (variables) e inicializadores de instancia es de acuerdo al orden en que se encuentran en el archivo. Luego de ello se ejecuta el constructor, pese a que el constructor pueda estar declarado antes que la declaración de variables o inicializadores de instancia.

## Distinguir entre Referencias de objetos y primitivos.

* Java tiene 8 tipos primitivos: boolean, byte, short, int, long, float, double, char.

|  |  |
| --- | --- |
| **TIPO** | **BITS** |
| boolean | True/false |
| byte | 8-bit inegral value |
| short | 16-bit inegral value |
| int | 32-bit inegral value |
| long | 64-bit inegral value |
| float | 32-bit floating point value |
| double | 64-bit floating point value |
| char | 16-bit Unicode valie |

* Float requiere un sufijo F seguido del número.
* Los tipos numéricos tienen la siguiente cantidad de números: byte (2^8), short (2^16), int (2^32), long (2^64), considerando los negativos y considerando al cero como positivo se tendría por ejemplo para byte el rango de -128 – 127.
* Otro punto que debe conocerse es que cuando un número se presenta como tal en el código es llamado un literal, de forma predeterminada Java asume que el literal es un int. Por ejemplo

**long** max = 3123456789; // DOES NOT COMPILE

**long** max = 3123456789L; // COMPILE

* Tambien pueden utilizarse otras bases númericas: octal con prefijo 0 (017), hexadecimal con prefijo 0x (0xFF), binario con prefijo 0b (0b10).
* La ultima cosa que se debe saber es que desde Java 7 se permitió agregar guiones bajos a los números para un mejor entendimiento.

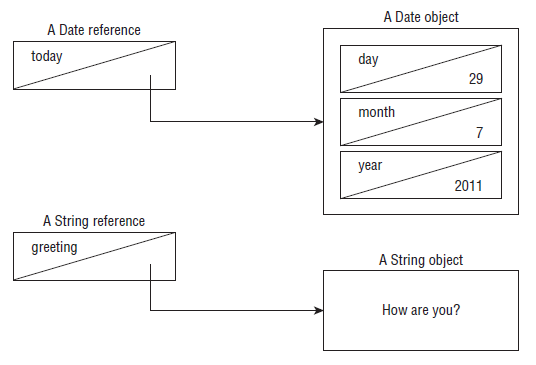
**double** notAtStart = \_1000.00; // DOES NOT COMPILE

**double** notAtEnd = 1000.00\_; // DOES NOT COMPILE

**double** notByDecimal = 1000\_.00; // DOES NOT COMPILE

**double** annoyingButLegal = 1\_00\_0.0\_0; // this one compiles

* A diferencia de los primitivos que guardan el valor, las variables de referencia solo apuntan al objeto guardado en memoria guardan la dirección de memoria donde el objeto está guardado (Java no permite conocer la dirección de memoria a diferencia de otros lenguajes como C por ejemplo).



* Existen algunas diferencias importantes entre primitivos y referencias:
  + A una variable de referencia se le puede asignar null pero a un primitivo no, en la compilación se mostraría con error.
  + Las variables de referencia pueden usarse para llamar métodos cuando no apuntan a null, los primitivos no tiene métodos a los que llamar.
  + Primitivos comienzan con minúscula, mientras que todas las clases comienzan con mayúscula, siendo una convención que debería seguirse.

## Declarando e Inicializando Variables

* Ya que esta parte es sencilla, solo coloco algunos ejemplos con posibles trucos engañosos que podrían venir en el examen.

**boolean** b1**,** b2**;** //BIEN

String s1 **=** "1"**,** s2**;** //BIEN

**double** d1**,** **double** d2**;** //MAL (el tipo debe colocarse solo una vez)

**int** i1**;** **int** i2**;** //BIEN (el ; separa sentencias)

**int** i3**;** i4**;** //MAL (i4 no tiene tipo)

* Lo bueno de Java es que se utilizan las mismas reglas para crear ya sea un nombre de variable, método, clase, y todo lo que se puede nombrar. Solo hay 3 reglas que deben seguirse:
  + El nombre debe iniciar con una letra, con $ o \_.
  + Los caracteres siguientes pueden ser números además de letras, $ o \_.
  + No se pueden usar las palabras reservadas por Java para nombrar algo, pero como Java es sensible a mayúsculas y minúsculas podrían generarse nombres con esa excepción (Igual no lo hagan por favor).

okidentifier //legal

$OK2Identifier //legal

\_alsoOK1d3ntifi3r //legal

\_\_SStillOkbutKnotsonice$ //legal

3DPointClass // identificadores no pueden iniciar con un número

hollywood@vine // @ no es una letra, dígito, $ o \_

**\***$coffee // \* no es una letra, dígito, $ o \_

**public** // public es una palabra reservada

## Inicialización Predeterminada de Variables

* Las variables locales son las que se declaran dentro de los métodos, estas deben ser inicializadas siempre para ser utilizadas. Si se utiliza una variable declarada pero no inicializada saltará un error de compilación.
* Las variables de instancia y variables de clase si tienen una inicialización predeterminada (recordar que las variables de clase son aquellas declaradas como estáticas, se verá más en el Capítulo 4). No es necesario memorizar el predeterminado de char para el examen.

|  |  |
| --- | --- |
| **TIPO DE VARIABLE** | **INICIALIZACION PREDETERMINADA** |
| boolean | false |
| byte, short, int, long | 0 |
| float, double | 0.0 |
| char | ‘\u0000’ (NUL) |
| Todas las referencias a objetos | null |

## Ámbitos de las variables (Variable Scope)

* Las variables locales pueden declararse dentro del método o como parámetro del método.
* Las variables locales no pueden tener un ámbito mayor al del método al que pertenecen pero si pueden tener un ámbito menor. Si una variable es por ejemplo declarada dentro de un if en el método, no podrá ser utilizada fuera del if pese a estar en el mismo método.

**public** **void** eatIfHungry**(boolean** hungry**)** **{**

**if** **(**hungry**)** **{**

**int** bitesOfCheese **=** 1**;**

**{**

**boolean** teenyBit **=** **true;**

System**.**out**.**println**(**bitesOfCheese**);**

**}**

**}**

System**.**out**.**println**(**teenyBit**);** // NO COMPILA

**}**

* Revisemos las reglas del ámbito (scope)
  + Variables Locales, en ámbito desde la declaración hasta el fin del bloque.
  + Variables de Instancia, en ámbito desde su declaración hasta que el objeto sea recolectado como basura (garbage collected)
  + Variables de Clase, en ámbito hasta que el programa termine.

## Ordenando elementos en una clase

* El orden es el siguiente:

|  |  |  |  |
| --- | --- | --- | --- |
| **ELEMENTO** | **EJEMPLO** | **REQUERIDO** | **DONDE VA?** |
| Declaracion de Paquete | package abc; | NO | Primera línea en el archivo |
| Imports | import java.util.\*; | NO | Inmediatamente después del paquete |
| Declaracion de Clase | public class C | SI | Inmediatamente después del import. |
| Declaracion de Variables | int value; | NO | Donde sea dentro del ámbito clase |
| Declaracion de métodos | void method(); | NO | Donde sea dentro del ámbito clase. |

* Los comentarios pueden ir en cualquiera lado
* Puede haber más de una clase en el archivo pero solo una de ellas debe ser pública y debe ser la que comparte el nombre del archivo.

## Destruyendo Objetos

* Java tiene el Garbage Collector para buscar objetos que ya no son necesarios.
* Todos los objetos Java son almacenados en el heap de memoria del programa, dependiendo del entorno este heap puede ser algo grande, pero siempre hay un límite y en algún momento si tu aplicación instancia muchos objetos y los deja en el heap, esta caerá con un error “out of memory“.
* Hay muchos algoritmos que utiliza el Garbage Collector pero no vamos a ver ello, lo que si debemos saber para el examen es que existe el System.go() y que al invocarlo no garantiza que corra ya que lo que hace es sugerirle a Java que éste podría ser un buen momento para liberar el heap (Java puede ignorarte simplemente).
* Java tiene mucha paciencia y mantendrá el objeto en el heap hasta que ya no sea accesible, un objeto no es accesible por las siguientes causas:
  + El objeto no tiene ninguna variable de referencia apuntando a él (quedó al aire cuando ninguna variable ya le apunta).
  + Todas las referencias al objeto quedaron fuera de alcance (fuera de ámbito – scope).

**public** **class** Scope **{**

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

String one**,** two**;**

one **=** **new** String**(**"a"**);**

two **=** **new** String**(**"b"**);**

one **=** two**;** // primera casuistica (objeto "a") objeto sin referencias

String three **=** one**;**

one **=** **null;**

**}** // segunda casuistica (objeto "b") queda fuera de ámbito o alcance

**}**

* Existe un método finalize() en las clases que corre si el Garbage Collector se ejecuta, si no se ejecuta nunca corre y si se ejecuta y falla la recolección y luego vuelve a ejecutarse ya no se vuelve a ejecutar el finalize porque ya se ejecutó la primera vez. Esto es lo único que se tiene que saber para el examen: solo se ejecuta el método si el garbage collector corre y solo se ejecuta cero o una vez.

## Beneficios de Java

* Lenguaje orientado a objetos
* Encapsulamiento
* Independiente de la plataforma
* Robusto: una de las mayores ventajas de Java frente a C++ es que evita la pérdida de memoria (memory leak) ya que la misma JVM es quien maneja la memoria y el recolector de basura automaticamente.
* Simple: Java fue destinado a ser más simple que C++, por ello no utiliza punteros ni sobrecarga de operadores (a+b por ejemplo podía ser interpretado como casi cualquier cosa)
* Seguro: Java corre en la JVM, que en si es una caja de arena (sandbox) que hace complicado que el código Java haga cosas malas al Sistema donde corre.

# Capítulo 2: Operadores y Sentencias

* Tres tipos de operadores son utilizados en Java: unario, binario y ternario.
* No hay un orden estricto de izquierda a derecha para resolver operadores, hay operaciones que se evalúan antes de acuerdo a un nivel de precedencia, si dos operaciones tienen el mismo nivel, ahí recién Java opta por hacerlo de izquierda a derecha.

|  |  |
| --- | --- |
| **Operador** | **Símbolos y Ejemplos** |
| Post-unary operators | expression++, expression-- |
| Pre-unary operators | ++expression, --expression |
| Other unary operators | +, -, ! |
| Multiplication/Division/Modulus | \*, /, % |
| Addition/Subtraction | +, - |
| Shift operators | <<, >>, >>> |
| Relational operators | <, >, <=, >=, instanceof |
| Equal to/not equal to | ==, != |
| Logical operators | &, ^, | |
| Short-circuit logical operators | &&, || |
| Ternary operators | boolean expression ? expression1 : expression2 |
| Assignment operators | =, +=, -=, \*=, /=, %=, &=, ^=, !=, <<=, >>=, >>>= |

* Todos los operadores aritméticos pueden ser aplicados a todos los tipos primitivos excepto boolean y String. Además solo los operadores + y += pueden ser aplicados a un String.
* Para el examen no se pedirá que se calcule el módulo (%) de un número negativo ni un decimal.

## Promoción de Valores

Hay 4 reglas para la promoción de tipos de datos en valores de una operación.

1. Si dos valores tienen diferentes tipos de dato, Java convertirá automáticamente el valor tipo de dato menor al de tipo de dato mayor.
2. Si uno de los valores es entero y el otro de coma flotante, Java convertirá automáticamente el valor de tipo entero a un tipo de valor de coma flotante.
3. Tipos de dato pequeños como **byte, short y char** son promovidos a **int** apenas se estén usando con operadores aritméticos binarios, incluso si no hay ningún valor **int** en la operación.
4. Una vez que en todos los valores han sido promovidos sus tipos de dato entonces el resultado de la operación tendrá ese mismo tipo de dato al que se ha promovido.

*Revisar el código para los ejemplos de las reglas (ValuesPromotion.java).*

## Valores Unarios

* Como es posible percatarse, los operadores unarios **+** y **-** son para cambiar de signo al número y obviamente no pueden aplicarse a un **boolean** porque caería en error de compilación.
* Así mismo el símbolo **!** niega el valor Verdadero o Falso de un **boolean** y no puede aplicarse a otro tipo de dato (solo boolean).

**int** x **=** **!**5**;** // NO COMPILA

**boolean** y **=** **-true;** // NO COMPILA

**boolean** z **=** **!**0**;** // NO COMPILA

### Valores de Incremento y Decremento

* Se debe tener mucho cuidado en estos casos; si el operador está precediendo al número (preincremento y predecremento) entonces el operador es aplicado y el valor retornado a la expresión es el nuevo valor. Alternativamente, si el operador se encuentra luego del número (postincremento y postdecremento), entonces el valor original del número es retornado a la expresión, con el operador aplicado luego de que retorna el valor.

**int** counter **=** 0**;**

System**.**out**.**println**(**counter**);** // Outputs 0

System**.**out**.**println**(++**counter**);** // Outputs 1

System**.**out**.**println**(**counter**);** // Outputs 1

System**.**out**.**println**(**counter**--);** // Outputs 1

System**.**out**.**println**(**counter**);** // Outputs 0

* Una práctica común en los exámenes es colocar una expresión con varios incrementos y decrementos sobre una misma variable.

**int** x **=** 3**;**

**int** y **=** **++**x **\*** 5 **/** x**--** **+** **--**x**;**

System**.**out**.**println**(**"x is " **+** x**);**

System**.**out**.**println**(**"y is " **+** y**);**

**int** y **=** 4 **\*** 5 **/** x**--** **+** **--**x**;** // x assigned value of 4

**int** y **=** 4 **\*** 5 **/** 4 **+** **--**x**;** // x assigned value of 3

**int** y **=** 4 **\*** 5 **/** 4 **+** 2**;** // x assigned value of 2

x is 2

y is 7

## Valores binarios adicionales

### Operadores de Asignación

* El operador de asignación más simple es el =.
* Java automáticamente promueve de tipos de datos más pequeños a tipos de datos mayores.
* Saldrá una excepción de compilación si intenta asignarse a un tipo de dato pequeño un valor con tipo de dato mayor.

**int** x **=** 1.0**;** // NO COMPILA (asignar un doble a un int)

**short** y **=** 1921222**;** // NO COMPILA (el número sobrepasa el máximo valor de short)

**int** z **=** 9f**;** // NO COMPILA (asignar un float a un int)

**long** t **=** 192301398193810323**;** NO COMPILA (el valor necesita un L para que sea long)

* Para que los ejemplos anteriores compilen tendría que agregarse un casteo explícito. Pero esto trae un costo, si el valor casteado sobrepasa al máximo valor de la variable a asignar, al valor se le coloca el siguiente valor menor (ejemplo 2 **y = 20678**). Hay algo análogo para cuando el número es demasiado menor para calzar en el tipo.

**short** x **=** 10**;**

**short** y **=** 3**;**

**short** z **=** x **\*** y**;** // NO COMPILA

**short** z **=** **(short)(**x **\*** y**);** //SI COMPILA

### Operadores de Asignación Compuestos

* Para el examen solo es necesario conocer los operadores += y -=.

**int** x **=** 2**,** z **=** 3**;**

x **=** x **\*** z**;** // Simple assignment operator

x **\*=** z**;** // Compound assignment operator

* Como se puede observar en el ejemplo siguiente, la prima operación no compila debido a que se quiere asignar un valor de tipo de dato mayor (**long**) a una variable de tipo de dato menor (**int**). Una ventaja de los operadores de asignación compuestos es que esta misma operación sin compilará ya que luego de hacer la operación el casteo es automático al tipo de la variable a quien se asignará el resultado.

**long** x **=** 10**;**

**int** y **=** 5**;**

y **=** y **\*** x**;** // DOES NOT COMPILE

y **\*=** x**;** // SI COMPILA

* Otro ejemplo válido es el siguiente donde se demuestra que una asignación en si es una expresión, permitiendo esta fea pero válida operación.

**long** x **=** 5**;**

**long** y **=** **(**x**=**3**);**

System**.**out**.**println**(**x**);** // Outputs 3

System**.**out**.**println**(**y**);** // Also, outputs 3

### Operadores Relacionales

* Estos operadores son exclusivamente para números, si ambos valores comparados tienen diferente tipo de dato al igual que en lo visto previamente el de tipo de dato menor se promueve al tipo de dato del operando de tipo de dato mayor.

**int** x **=** 10**,** y **=** 20**,** z **=** 10**;**

System**.**out**.**println**(**x **<** y**);** // Outputs true

System**.**out**.**println**(**x **<=** y**);** // Outputs true

System**.**out**.**println**(**x **>=** z**);** // Outputs true

System**.**out**.**println**(**x **>** z**);** // Outputs false

* En el ultimo ejemplo se da falso debido a que tanto el operador > y < solo da verdadero cuando es estrictamente mayor o menor, en este caso es igual así que arroja falso.

### Operadores Lógicos