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

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

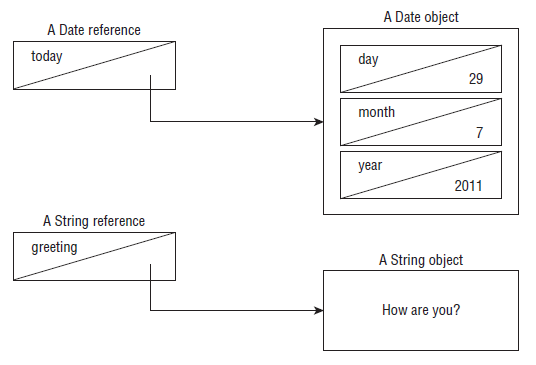
**double** notAtStart = \_1000.00; // DOES NOT COMPILE

**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 primera 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 si 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 último 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

* Los operadores lógicos (&), (|) y (^) pueden aplicarse tanto a tipos de datos numéricos como booleanos, pero para el examen no se considerará la comparación numérica.
* La tabla de verdad para estos operadores es conocida, con estos tips es fácil recordarla:
  + Y es solo verdad si ambos operandos son verdad.
  + O inclusivo es falso solamente si ambos operandos son falsos.
  + O exclusivo es verdad solamente si ambos operandos son diferentes.
* Luego tenemos los operadores lógicos && y || que generalmente son referidos como operadores lógicos de corto circuito, son prácticamente iguales que los operadores anteriores salvo que el lado derecho de la expresión no siempre es evaluado si la evaluación del lado izquierdo de la expresión basta para encontrar la respuesta.

**boolean** x **=** **true** **||** **(**y **<** 4**);**

* En el ejemplo anterior por ejemplo, para el operador O se cumple que solo es falso si ambas expresiones son falsas, entonces si se sabe que la primera expresión (x=true) es verdadera ya no sería necesario evaluar la segunda expresión.
* Estos operadores son bien utilizados para cuando se desea evaluar que un objeto no sea nulo antes de evaluar la segunda expresión.

**if(**x **!=** **null** **&&** x**.**getValue**()** **<** 5**)** **{**

// Do something

**}**

**if(**x **!=** **null** **&** x**.**getValue**()** **<** 5**)** **{** // Throws an exception if x is null

// Do something

**}**

* El siguiente ejemplo es una pregunta tipo examen. En el ejemplo el valor de x que se muestra será 6 ya que al ser la primera expresión lógica Verdadera, ya no se evalúa la segunda expresión y la variable **x** nunca aumenta.

**int** x **=** 6**;**

**boolean** y **=** **(**x **>=** 6**)** **||** **(++**x **<=** 7**);**

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

### Operadores de Igualdad

* Determinar la igualdad en Java no es tan sencillo como parece, ya que hay una diferencia semántica entre “dos objetos son iguales” y “dos objetos son equivalentes”. Y se complica más ya que para los tipos primitivos numéricos y booleanos no existe esa distinción.
* Los operadores de igualdad (== y !=) son utilizados en tres escenarios:
  + Comparando dos tipos numéricos primitivos. Si uno de los valores numéricos es de diferente tipo de dato, el de tipo de dato menor se promueve al mismo tipo de dato del otro operando. Por ejemplo: 5 == 5.00 es verdadero.
  + Comparando dos valores booleanos.
  + Comparando dos objetos, incluyendo null y String.
* La comparación se limita a lo indicado arriba:

**boolean** x **=** **true** **==** 3**;** // NO COMPILA

**boolean** y **=** **false** **!=** "Giraffe"**;** // NO COMPILA

**boolean** z **=** 3 **==** "Kangaroo"**;** // NO COMPILA

* Al comparar objetos el operador de igualdad solo se aplica a la referencia de objetos, no a los objetos a los que apunta. Dos referencias son iguales siempre y cuando apunten al mismo objeto o ambos apunten a null.

File x **=** **new** File**(**"myFile.txt"**);**

File y **=** **new** File**(**"myFile.txt"**);**

File z **=** x**;**

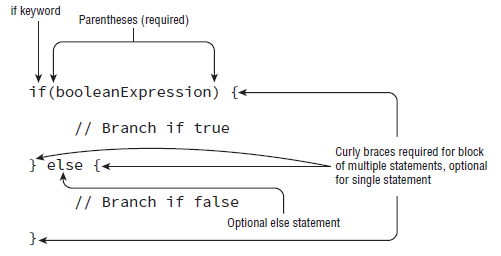
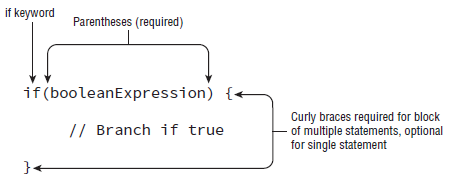
System**.**out**.**println**(**x **==** y**);** // Outputs false

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

* En el examen pueden venir preguntas con objetos de clases desconocidas que aparentemente complican la pregunta pero cuando en realidad solo deberían revisarse los operadores.

## Tipos de declaraciones o sentencias

### Declaración If Then and Else



* Pueden omitirse las llaves para cuando solo se tiene una expresión luego del if, aunque es buena práctica en Java utilizar las llaves para un mejor orden.

**if(**hourOfDay **<** 11**)**

System**.**out**.**println**(**"Good Morning"**);**

morningGreetingCount**++;**

* En el cuadro anterior por ejemplo el contador *morningGreetingCount* se incrementaría aunque el valor de *hourOfDay* sea mayor o igual a 11 (no entraría en la condicional).
* En los casos donde se anidan varios bloques If, recordar que cada bloque es excluyente, solo se ejecutará el primer bloque evaluado que cumpla la condición. En el ejemplo siguiente si *hourOfDay* es 5 por ejemplo solo entraría al primer bloque.

**if(**hourOfDay **<** 15**)** **{**

System**.**out**.**println**(**"Good Afternoon"**);**

**}** **else** **if(**hourOfDay **<** 11**)** **{**

System**.**out**.**println**(**"Good Morning"**);** // UNREACHABLE CODE

**}** **else** **{**

System**.**out**.**println**(**"Good Evening"**);**

**}**

* Recordar también que dentro de los paréntesis del ***If*** solo se evalúan booleanos, ninguna otro tipo de dato primitivo.

**int** x **=** 1**;**

**if(**x**)** **{** // NO COMPILA

**...**

**}**

**if(**x **=** 5**)** **{** // NO COMPILA

**...**

**}**

#### Operador Ternario

* Es una forma abreviada del ***If Then***.
* Se recomienda utilizar paréntesis en los bloques para un mejor orden pero no es necesario.
* La segunda y tercer expresión podrían ser de distintos tipos pero se debe tener cuidado si hay un operador de asignación también involucrado.

System**.**out**.**println**((**y **>** 5**)** **?** 21 **:** "Zebra"**);**

**int** animal **=** **(**y **<** 91**)** **?** 9 **:** "Horse"**;** // NO COMPILA

* Recordar también que solo se ejecuta la expresión que cumple la condición, en el examen pueden venir preguntas que intenten confundir con ello. En el ejemplo debajo el valor de z nunca se ejecuta y por ello no se incrementa su valor.

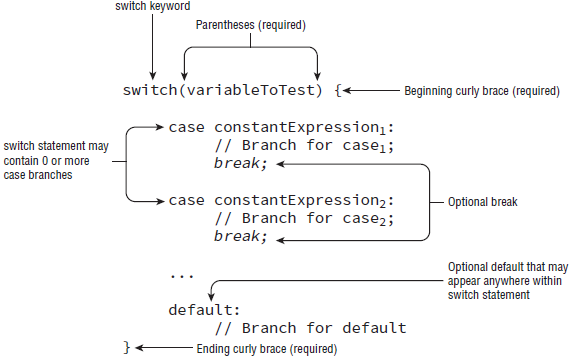
**int** y **=** 1**;**

**int** z **=** 1**;**

**final** **int** x **=** y**<**10 **?** y**++** **:** z**++;**

System**.**out**.**println**(**y**+**","**+**z**);** // Salida 2,1

### Declaración Switch



* Es una compleja estructura de toma de decisiones donde una variable es evaluada. Luego de que se evalúa la variable el flujo de ejecución se direcciona a la primera rama que coincida (case). Si no se encuentra se irá hacia la rama default. Si no existe un default, la ejecución del switch termina.
* Los tipos soportados actualmente por Switch son (memorizar la lista ya que por ejemplo long ni Long son soportados):
  + int e Integer
  + byte y Byte
  + short y Short
  + char y Character
  + String
  + Valores Enum
* Como se ha comentado, la sentencia break es opcional; si todas las ramas tienen un break al final no habría problema y el flujo de ejecución iría bien al Case que coincide con el valor o con el default si existe. Tener en cuenta que el orden del default no importa, y siempre y cuando encuentre una coincidencia en un Case, el flujo irá a ese Case aunque el default este líneas arriba.
* En el caso de que no se coloque break, el flujo irá hacia el Case que coincida con el valor actual de la variable y a partir de ahí continuará entrando a los Case que se encuentren debajo hasta que encuentre un break o termine el Switch. Esto también aplica si no encuentra coincidencia y entra al default, pero luego del default hay ramas Case declaradas, sin el break también recorrerá todos los Case debajo hasta que encuentre un break.

**int** dayOfWeek **=** 5**;**

**switch(**dayOfWeek**)** **{**

**case** 0**:**

System**.**out**.**println**(**"Sunday"**);**

**default:**

System**.**out**.**println**(**"Weekday"**);**

**case** 6**:**

System**.**out**.**println**(**"Saturday"**);**

**break;**

**}**

* En el ejemplo anterior por ejemplo, la salida será: Weekday y Saturday
* Si se cambia la variable dayOfWeek a 6 la salida será: Saturday.
* Si se cambia la variable dayOfWeek a 0 la salida será: Sunday, Weekday y Saturday.
* A los que elaboran el examen les gusta jugar con estos ejemplo, tener cuidado.
* Un dato importante es que los tipos de datos colocados en las ramas Case deben ser del mismo tipo que el de la variable a evaluar en el Switch, sino no compilará.
* Recordar también que en el Case solo se puede colocar literales, constantes **enum** o constantes **final**.

**private** **int** getSortOrder**(**String firstName**,** **final** String lastName**)** **{**

String middleName **=** "Patricia"**;**

**final** String suffix **=** "JR"**;**

**int** id **=** 0**;**

**switch(**firstName**)** **{**

**case** "Test"**:**

**return** 52**;** // SI COMPILA YA QUE EL RETURN AL IGUAL QUE EL BREAK TAMBIEN SIRVE PARA FINALIZAR LA SENTENCIA SWITCH PREMATURAMENTE.

**case** middleName**:** // NO COMPILA PORQUE NO ES UN LITERAL

id **=** 5**;**

**break;**

**case** suffix**:** // SI COMPILA

id **=** 0**;**

**break;**

**case** lastName**:** // NO COMPILA, YA QUE PESE A SER UN FINAL NO ES CONSTANTE

id **=** 8**;**

**break;**

**case** 5**:** // NO COMPILA

id **=** 7**;**

**break;**

**case** 'J'**:** // NO COMPILA

id **=** 10**;**

**break;**

**case** java**.**time**.**DayOfWeek**.**SUNDAY**:** // NO COMPILA PESE A SER UN ENUM PORQUE EL TIPO DE DATO NO ES EL MISMO QUE LA VARIABLE.

id**=**15**;**

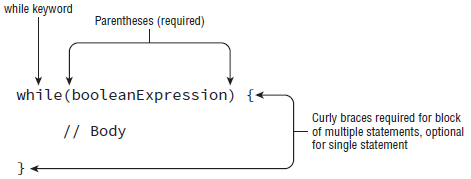
**break;**

**}**

**return** id**;**

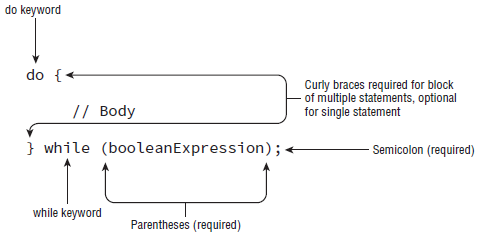
**}**

### Declaración While



* Es una estructura de control de repeticiones (loop).
* Durante la ejecución la expresión es evaluada antes de cada iteración. Es importante comprender que el loop puede terminar después de la primera evaluación sin entrar al bloque de código dentro del while.
* Debe tenerse cuidado con los loops infinitos, asegurándonos de que al programar un loop siempre se llegue a una condición que le permita terminar.

### Declaración Do-While



* La diferencia con el while es que en este caso el bloque de código se ejecutará al menos una vez ya que la validación no se ejecuta primero como en el while.

**while(**x **>** 10**)** **{**

x**--;**

**}**

**if(**x **>** 10**)** **{**

**do** **{**

x**--;**

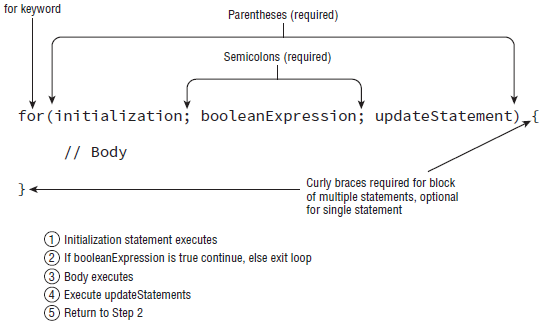
**}** **while(**x **>** 10**);**

**}**

* El caso anterior nos muestra que es posible convertir un do while en un while agregando un if, por lo que no hay mucha diferencia entre usar uno u otro.

### Declaración For

#### Declaración for básica



* Como se observa en la imagen, la sentencia for tiene 3 bloques dentro de los paréntesis, los 3 bloques no son obligatorios, podrían ir vacios, así como también es permitido colocar varias expresiones en los 3 bloques (en inicialización y updateStatement separados por comas y en booleanExpression separados por operadores lógicos).
* Es posible crear un loop infinito con la siguiente expresión: **for( ; ; ) {**
* Aquí algunos ejemplos:

**int** x **=** 0**;**

**for(long** y **=** 0**,** z **=** 4**;** x **<** 5 **&&** y **<** 10**;** x**++,** y**++)** **{**

System**.**out**.**print**(**y **+** " "**);**

**}**

System**.**out**.**print**(**x**);**

**int** x **=** 0**;**

**for(long** y **=** 0**,** x **=** 4**;** x **<** 5 **&&** y **<** 10**;** x**++,** y**++)** **{** // NO COMPILA, variable X ya declarada

System**.**out**.**print**(**x **+** " "**);**

**}**

**int** x **=** 0**;**

**long** y **=** 10**;**

**for(**y **=** 0**,** x **=** 4**;** x **<** 5 **&&** y **<** 10**;** x**++,** y**++)** **{** //AQUI SI COMPILA

System**.**out**.**print**(**x **+** " "**);**

**}**

// NO COMPILA, bloque inicializacion solo debe tener un tipo

**for(long** y **=** 0**,** **int** x **=** 4**;** x **<** 5 **&&** y**<**10**;** x**++,** y**++)** **{**

System**.**out**.**print**(**x **+** " "**);**

**}**

**for(long** y **=** 0**,** x **=** 4**;** x **<** 5 **&&** y **<** 10**;** x**++,** y**++)** **{**

System**.**out**.**print**(**y **+** " "**);**

**}**

// NO COMPILA, variable X solo entra en el ámbito del for

System**.**out**.**print**(**x**);**

#### Declaración for-each

#### 

* Se implementó en Java 5, se compone de una sección de declaración y la otra de un objeto a ser iterado. El lado derecho debe constar de un **Java** **array** o un objeto que implemente la interfaz **java.lang.Iterable**, lo cual incluye la mayoría de elementos de Java Collections. El lado izquierdo debe contener una declaración de una variable de instancia y su tipo debe coincidir con el tipo de los objetos miembros del objeto del lado derecho (incluye herencia). En cada iteración se le asigna a la variable declarada en el lado izquierdo un elemento del array o la colección del lado derecho (Para el examen solo se toma en cuenta a List y ArrayList para lo que son colecciones).

String names **=** "Lisa"**;**

**for(**String name **:** names**)** **{** // NO COMPILA, names no es array ni Iterable

System**.**out**.**print**(**name **+** " "**);**

**}**

String**[]** names **=** **new** String**[**3**];**

**for(int** name **:** names**)** **{** // NO COMPILA, variable name de tipo incorrecto.

System**.**out**.**print**(**name **+** " "**);**

**}**

## Control de Flujos Avanzado

* Se pueden dar bucles anidados, por ejemplo para matrices multidimensionales.
* Pueden utilizarse labels (etiquetas) para un mejor ordenamientos de loops (buenas practicas), su formato es el mismo que para los identificadores; se acostumbra colocarlos en mayúsculas con un guion abajo entre palabras. Luego del label se colocan dos puntos y ahí recién comienza la definición del loop.

**int[][]** myComplexArray **=** **{{**5**,**2**,**1**,**3**},{**3**,**9**,**8**,**9**},{**5**,**7**,**12**,**7**}};**

OUTER\_LOOP**:** **for(int[]** mySimpleArray **:** myComplexArray**)** **{**

INNER\_LOOP**:** **for(int** i**=**0**;** i**<**mySimpleArray**.**length**;** i**++)** **{**

System**.**out**.**print**(**mySimpleArray**[**i**]+**"\t"**);**

**}**

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

**}**

* La sentencia break como se ha visto en ejemplos anteriores para switch se utiliza para transferir el control del flujo de ejecución fuera la sentencia anidada (entre llaves, switch, for, while, etc). Si se tiene loops anidados el break se aplicará al loop más cercano, a menos que se utilice un label, así podría terminar un loop de más alto nivel.

**public** **class** SearchSample **{**

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

**int[][]** list **=** **{{**1**,**13**,**5**},{**1**,**2**,**5**},{**2**,**7**,**2**}};**

**int** searchValue **=** 2**;**

**int** positionX **=** **-**1**;**

**int** positionY **=** **-**1**;**

PARENT\_LOOP**:** **for(int** i**=**0**;** i**<**list**.**length**;** i**++)** **{**

**for(int** j**=**0**;** j**<**list**[**i**].**length**;** j**++)** **{**

**if(**list**[**i**][**j**]==**searchValue**)** **{**

positionX **=** i**;**

positionY **=** j**;**

**break** PARENT\_LOOP**;**

**}**

**}**

**}**

**if(**positionX**==-**1 **||** positionY**==-**1**)** **{**

System**.**out**.**println**(**"Value "**+**searchValue**+**" not found"**);**

**}** **else** **{**

System**.**out**.**println**(**"Value "**+**searchValue**+**" found at: " **+**

"("**+**positionX**+**","**+**positionY**+**")"**);**

**}**

**}**

**}**

* En el código anterior por ejemplo el **break** termina la ejecución del loop principal cuando encuentra el valor 2 en la posición (1,1). Si no se colocará el label en el break, el loop que finalizará sería el inmediato que está ejecutándose en ese momento (segundo).
* La sentencia **continue** se aplica de manera similar que el break, con la diferencia de que en lugar de finalizar la ejecución del loop, el continue termina el flujo y regresa a la condición booleana del loop para ser evaluada y continuar el loop.

**public** **class** SwitchSample **{**

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

FIRST\_CHAR\_LOOP**:** **for** **(int** a **=** 1**;** a **<=** 4**;** a**++)** **{**

**for** **(char** x **=** 'a'**;** x **<=** 'c'**;** x**++)** **{**

**if** **(**a **==** 2 **||** x **==** 'b'**)**

**continue** FIRST\_CHAR\_LOOP**;**

System**.**out**.**print**(**" " **+** a **+** x**);**

**}**

**}**

**}**

**}**

* En el ejemplo anterior el resultado sería 1a, 3a, 4a.
* Si removiésemos el label del continue el resultado sería: 1a, 1c, 3a, 3c, 4a, 4c.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **permite labels** | **permite break** | **permite continue** |
| **if** | si | no | no |
| **while** | si | si | si |
| **do while** | si | si | si |
| **for** | si | si | si |
| **switch** | si | si | no |

# Capítulo 3: Core Java APIs

## Creando y manipulando Strings

* La clase String no necesita un new para inicializarse, pero puede hacerse.
* Es inmutable, es decir que no puede modificarse una vez inicializada (recordar que una clase es inmutable cuando no tiene setters, solo getters por ello sus elementos solo se conservan con lo inicializado en el constructor. En Java las clases inmutables son **final**).

String s1 **=** "1"**;**

String s2 **=** s1**.**concat**(**"2"**);**

s2**.**concat**(**"3"**);**

System**.**out**.**println**(**s2**);**

* En el ejemplo anterior el resultado de s2 es “12”, ya que la clase String es inmutable. Si se le asigna otro texto a la variable de referencia s1, por ejemplo “5” entonces ya haría referencia a otro objeto, no a “1”.
* El String pool se encuentra dentro de la JVM y se utiliza para gestionar todos los Strings literales de la aplicación ya que los Strings son los elementos más utilizados en una aplicación.
* Todo String entra al String pool excepto los inicializados con **new**.
* Los Strings dentro del String pool son más eficientes que los que se encuentran fuera (no literales). myObject.toString() es un String pero no un literal por ejemplo y quedaría fuera del String pool.
* Los String que no están en el String pool son recolectados por el Garbage Collector como cualquier objeto.

### Métodos Importantes de String

#### **lenght()**

Retorna la cantidad de caracteres del String.

#### **charAt()**

Encuentra un carácter en la posición especifica indicada (*char charAt(int index)*).

#### **indexOf()**

Busca los caracteres indicados en el String y devuelve la posición de la primera coincidencia. En la opción con parámetro fromIndex, comienza a buscar desde esa posición.

**int** indexOf**(char** ch**)**

**int** indexOf**(char** ch**,** **int** fromIndex**)**

**int** indexOf**(**String str**)**

**int** indexOf**(**String str**,** **int** fromIndex**)**

#### **substring()**

Este método busca dentro del String de acuerdo a las posiciones indicadas en los parámetros (inicio, fin). En este caso por ejemplo si se deseara buscar hasta el final pero colocando el segundo parámetro (opcional) podría colocarse un número más a la posición final del String, sería redundante ya que no sería necesario el segundo parámetro pero igual no arrojaría error.

**int** substring**(int** beginIndex**)**

**int** substring**(int** beginIndex**,** **int** endIndex**)**

String string **=** "animals"**;**

string**.**substring**(**3**,** 7**);** //no sería necesario el segundo parámetro.

System**.**out**.**println**(**string**.**substring**(**3**,** 3**));** // empty string

System**.**out**.**println**(**string**.**substring**(**3**,** 2**));** // throws exception

System**.**out**.**println**(**string**.**substring**(**3**,** 8**));** // throws exception

#### **toUpperCase() and toLowerCase()**

No es necesaria mucha explicación, su propio nombre lo dice.

#### **equals() and equalsIgnoreCase()**

Compara cadenas de String si son iguales, en el segundo caso sin tomar en cuenta mayúscula o minúscula.

#### **startsWith() and endsWith()**

Devuelve true o false si encuentra el String del parámetro en la cadena (diferencia mayúscula de minúscula).

System**.**out**.**println**(**"abc"**.**startsWith**(**"a"**));** // true

System**.**out**.**println**(**"abc"**.**startsWith**(**"A"**));** // false

System**.**out**.**println**(**"abc"**.**endsWith**(**"c"**));** // true

System**.**out**.**println**(**"abc"**.**endsWith**(**"a"**));** // false

#### **contains()**

Este método busca coincidencias en el String. La coincidencia puede estar en cualquier lugar de la cadena (también toma en cuenta las mayúsculas y minúsculas).

#### **replace()**

Hace un reemplazo simple en el String. Puede reemplazar un **char** o un **CharSequence**. CharSequence es una interfaz que puede representar muchas clases, como String o StringBuilder por ejemplo.

System**.**out**.**println**(**"abcabc"**.**replace**(**'a'**,** 'A'**));** // AbcAbc //Char parameter

System**.**out**.**println**(**"abcabc"**.**replace**(**"a"**,** "A"**));** // AbcAbc //String parameter

#### **trim()**

Elimina los espacios en blanco del inicio y el final, esto incluye a las tabulaciones, saltos de línea y retornos de carro (\t, \n, \r)

* Los métodos de String pueden encadenarse, y en cada ejecución de un método se devuelve un nuevo objeto String hasta que al final se le asigna a la variable (por ser inmutable):

String result **=** "AniMaL "**.**trim**().**toLowerCase**().**replace**(**'a'**,** 'A'**);**

## Utilizando StringBuilder

* A diferencia de String la clase StringBuilder es no inmutable, por ello puede alterarse su valor sin que se tenga que instanciar un nuevo objeto.

10**:** String alpha **=** ""**;**

11**:** **for(char** current **=** 'a'**;** current **<=** 'z'**;** current**++)**

12**:** alpha **+=** current**;**

13**:** System**.**out**.**println**(**alpha**);**

15**:** StringBuilder alpha **=** **new** StringBuilder**();**

16**:** **for(char** current **=** 'a'**;** current **<=** 'z'**;** current**++)**

17**:** alpha**.**append**(**current**);**

18**:** System**.**out**.**println**(**alpha**);**

* En el ejemplo anterior se aprecia lo indicado al inicio, para String se han tenido que instanciar 27 objetos al recorrer todo el abecedario, pero para StringBuilder se ha trabajado sobre el mismo objeto.
* Debido a que StringBuilder es no inmutable, puede alterarse su valor y en el examen no pondrán las cosas fáciles, como en el ejemplo siguiente, el resultado tanto para a y b será el mismo ya que ambos apuntan al mismo objeto.

4**:** StringBuilder a **=** **new** StringBuilder**(**"abc"**);**

5**:** StringBuilder b **=** a**.**append**(**"de"**);**

6**:** b **=** b**.**append**(**"f"**).**append**(**"g"**);**

7**:** System**.**out**.**println**(**"a=" **+** a**);**

8**:** System**.**out**.**println**(**"b=" **+** b**);**

* Hay tres formas de crear un StringBuilder:
  + StringBuilder sb1 = new StringBuilder();
  + StringBuilder sb2 = new StringBuilder("animal");
  + StringBuilder sb3 = new StringBuilder(10);
* La primera forma crea un StringBuilder vacío, la segunda uno con data y el tercero indica el espacio reservado de caracteres (capacidad).
* La capacidad predeterminada es 16, pero si se le asigna un valor no pasa nada ya que si se sobrepasa esa capacidad añadiendo caracteres al StringBuilder, Java le asigna más capacidad de forma automática.

### Métodos importantes de StringBuilder

#### **charAt(), indexOf(), length(), and substring()**

Funcionan igual que en String. Notar que substring devuelve String.

StringBuilder sb **=** **new** StringBuilder**(**"animals"**);**

String sub **=** sb**.**substring**(**sb**.**indexOf**(**"a"**),** sb**.**indexOf**(**"al"**));**

**int** len **=** sb**.**length**();**

**char** ch **=** sb**.**charAt**(**6**);**

System**.**out**.**println**(**sub **+** " " **+** len **+** " " **+** ch**);**

#### **append()**

Este método hace justo lo que dice, añade el valor del parámetro al StringBuilder y devuelve una referencia al actual StringBuilder, tiene muchas formas de acuerdo al tipo de parámetro que se le pase (no se necesita estar convirtiendo a String los valores a añadir).

#### **insert()**

Este método inserta un valor en determinada posición.

3**:** StringBuilder sb **=** **new** StringBuilder**(**"animals"**);**

4**:** sb**.**insert**(**7**,** "-"**);** // sb = animals-

5**:** sb**.**insert**(**0**,** "-"**);** // sb = -animals-

6**:** sb**.**insert**(**4**,** "-"**);** // sb = -ani-mals

7**:** System**.**out**.**println**(**sb**);**

#### **delete() and deleteCharAt()**

Este método delete es opuesto al insert, elimina a partir de una posición inicial (inclusiva) hasta otra posición final (exclusiva). El método deleteCharAt es para eliminar un solo carácter. Recordar que las posiciones inician en cero.

StringBuilder sb **=** **new** StringBuilder**(**"abcdef"**);**

sb**.**delete**(**1**,** 3**);** // sb = adef

sb**.**deleteCharAt**(**5**);** // throws an exception

#### **reverse()**

Este método hace de igual forma lo que dice, revierte la secuencia de los caracteres.

#### **toString()**

Convierte el valor a String.

* La diferencia entre StringBuilder (desde Java 5) con StringBuffer es que StringBuffer es thread safe (cuando puede accederse desde varios hilos sin causar problemas, “syncronized”), por ello es más lento.

### Entendiendo la Igualdad

* Lo visto anteriormente al comparar variables de referencia con == aplica para StringBuilder.

StringBuilder one **=** **new** StringBuilder**();**

StringBuilder two **=** **new** StringBuilder**();**

StringBuilder three **=** one**.**append**(**"a"**);**

System**.**out**.**println**(**one **==** two**);** // false

System**.**out**.**println**(**one **==** three**);** // true

* Veamos algo más complejo, sabiendo que en el String pool se almacena un solo String al que apuntan varias variables de referencia, lo que explica la igualdad del primer caso del ejemplo siguiente:

String x **=** "Hello World"**;**

String y **=** "Hello World"**;**

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

String x **=** "Hello World"**;**

String z **=** " Hello World"**.**trim**();**

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

String x **=** **new** String**(**"Hello World"**);**

String y **=** "Hello World"**;**

System**.**out**.**println**(**x **==** y**);** // false

* En el segundo caso es diferente debido a que el valor de z se calcula en tiempo de ejecución y no en tiempo de compilación, por ello no solo existe un objeto en el String pool al que apuntan ambas variables.
* En resumen, llamar a == en Strings compara que ambos apunten al mismo String en el String pool, hacerlo sobre un StringBuilder verifica que apunten al mismo objeto. Con equals en Strings se verifica la secuencia de la cadena de caracteres, equals sobre un StringBuilder también verifica que apunten al mismo objeto.
* La lección es clara, nunca usar == para comparar Strings, se debe utilizar el método **equals**.

## Entendiendo Arreglos en Java

* Un arreglo es un área de memoria en el heap.
* Un String y un StringBuilder son una especie de array, pero a diferencia de ellos pueden utilizarse cualquier otro tipo de dato.
* Pese a que un array pueda ser de tipo primitivo, siempre es una variable de referencia.
* Un array de tipo int por ejemplo siempre se instancia con su valor prederminado en cada espacio del array, en este caso ceros.

**int[]** numbers1 **=** **new** **int[**3**];** //array con ceros en sus 3 espacios

**int[]** numbers2 **=** **new** **int[]** **{**42**,** 55**,** 99**};** //array con 3 elementos

**int[]** numbers2 **=** **{**42**,** 55**,** 99**};** //array anonimo, igual al anterior

* Los corchetes de un array pueden estar antes del nombre de la variable o después, también pueden escribirse con espacios entre ellos y el tipo:

**int[]** numAnimals**;**

**int** **[]** numAnimals2**;**

**int** numAnimals3**[];**

**int** numAnimals4 **[];**

* Un caso particular en la declaración de variables es la siguiente:

**int[]** ids**,** types**;** //dos arrays de tipo int

**int** ids**[],** types**;** //un array de tipo int y una variable de tipo int

* Se debe de tener cuidado con los elementos que se le asignan a los arrays, deben ser del mismo tipo con el que se instanciaron o uno menos específico, por ejemplo un array de Object podría guardar cualquier tipo. Se debe tener cuidado también con un caso como el siguiente:

String**[]** strings **=** **{** "stringValue" **};**

Object**[]** objects **=** strings**;**

String**[]** againStrings **=** **(**String**[])** objects**;**

againStrings**[**0**]** **=** **new** StringBuilder**();** // NO COMPILA (diferente tipo)

objects**[**0**]** **=** **new** StringBuilder**();** // error de ejecución no de compilación!

* El método lenght para un array no considera si hay un valor en cada slot, sino que considera la cantidad de slots total del array.
* El método sort de java.util.Arrays ordena lo que hay dentro de un array, tener en cuenta que el ordenamiento de números es diferente al de Strings, un array con estos elementos { "10", "9", "100" } será ordenado como 10, 100, 9 ya que el ordenamiento de Strings es alfabético.
* Para la búsqueda se utiliza el método binarySearch de Arrays, pero para buscar la lista de elementos debe de estar ordenada. Si el elemento no se encuentra se arroja un número negativo que es la suma del negativo de la posición donde debería ir el elemento buscado y -1.

**int[]** numbers **=** **{**2**,**4**,**6**,**8**};**

System**.**out**.**println**(**Arrays**.**binarySearch**(**numbers**,** 2**));** // 0

System**.**out**.**println**(**Arrays**.**binarySearch**(**numbers**,** 4**));** // 1

System**.**out**.**println**(**Arrays**.**binarySearch**(**numbers**,** 1**));** // -1

System**.**out**.**println**(**Arrays**.**binarySearch**(**numbers**,** 3**));** // -2

System**.**out**.**println**(**Arrays**.**binarySearch**(**numbers**,** 9**));** // -5

* Cuando el array no está ordenado la búsqueda es impredecible, tener cuenta esto en el examen.
* Trabajando con arrays multidimensionales, se pueden inicializar con cantidades fijas para todos sus elementos o también puede inicializarse con cantidades variables

String [][] rectangle = **new** String[3][2];

**int**[][] differentSize = {{1, 4}, {3}, {9,8,7}}; *//cantidades diferentes al siguiente nivel*

**int** [][] args = **new** **int**[4][]; *//no se define el tamaño del siguiente nivel*

args[0] = **new** **int**[5];

args[1] = **new** **int**[3];

## Entendiendo un ArrayList

* Al igual que StringBuilder hay tres formas de inicializar un ArrayList.
  + ArrayList list1 = new ArrayList();
  + ArrayList list2 = new ArrayList(10);
  + ArrayList list3 = new ArrayList(list2);
* El primer caso crea un ArrayList con el espacio predeterminado para ingresar elementos pero no asigna nada, el segundo caso especifica el tamaño pero tampoco asigna nada (seguirá teniendo cero elementos), en el tercer caso se realiza una copia de ambos, el tamaño y el contenido.
* A partir de Java 5 se introdujo las clases genéricas, lo cual permite indicar qué clase se almacenará dentro del ArrayList. Asi también, desde Java 7 se permite omitir el tipo en el lado derecho.

ArrayList<String> list4 = **new** ArrayList<String>();

ArrayList<String> list5 = **new** ArrayList<>();

### Métodos importantes de ArrayList

#### **add()**

Inserta un nuevo elemento al ArrayList, tanto al final de la lista como indicando la posición. Si se hace la inserción indicando la posición entonces el elemento que ya se encontraba en esa posición y los que le siguen incrementarán en uno su posición. Para este método el retorno siempre será **true**.

**boolean** add(E element)

**void** add(**int** index, E element)

#### **remove()**

Remueve el primer elemento encontrado en la lista y retorna **true** y si no elimina nada retorna **false**; si se indica la posición remueve el elemento que se encuentra en esa posición y retorna el elemento eliminado.

**boolean** remove(Object object)

E remove(**int** index)

Si se indica una posición que no existe arrojara una excepción (IndexOutOfBoundsException). Existe un método removeIf pero utiliza expresiones Lambda, se verá en el siguiente capítulo.

#### **set()**

Reemplaza el elemento que se encuentra en la posición indicada y devuelve el elemento reemplazado.

E set(**int** index, E newElement)

#### **isEmpty() y size()**

Ambos métodos revisan cuantos slots han sido utilizados del ArrayList. Al igual que con el StringBuilder no se mide la capacidad sino la cantidad de slots utilizados en el momento.

**boolean** isEmpty()

**int** size()

#### **clear()**

Este método provee una manera fácil de limpiar el ArrayList de elementos, una vez utilizado el ArrayList volverá a estar vacío y de tamaño 0.

#### **contains()**

Este método revisa si el valor buscado se encuentra en el ArrayList. Este método llama a equals() en cada elemento del ArrayList.

**boolean** contains(Object object)

#### **equals()**

ArrayList implementa equals() por lo que puede compararse dos listas devolviendo true si ambas listas tienen el mismo tamaño y los mismos elementos en el mismo orden.

**boolean** equals(Object object)

### Clases Wrapper

* Todos los tipos primitivos tienen su clase Wrapper y por lo general es el mismo nombre pero con la primera letra en mayúscula, a excepción de int (Integer) y de char (Character).
* Las clases Wrapper tienen métodos que devuelven el primitivo, pero no hay que preocuparse mucho por trucos con métodos como intValue() por ejemplo, el **autoboxing** (ver siguiente sección) ha removido la necesidad de ellos.
* Hay métodos también para convertir un String a un primitivo o a una clase wrapper. El primer ejemplo debajo devuelve un primitivo mientras que el otro devuelve la clase Wrapper.

**int** primitive = Integer.parseInt("123");

Integer wrapper = Integer.valueOf("123");

* Si el String pasado como parámetro no es un número válido, devolverá la excepción NumberFormatException.
* **El Autoboxing se da desde Java 5**, gracias a ello no es necesario castear ni convertir de la clase Wrapper al primitivo o viceversa, Java lo hace automáticamente. Por ello es posible por ejemplo asignarle un int literal al Wrapper Integer (Integer num = 10).
* Se debe tener cuidado con asignarle null a una variable primitiva.

List<Integer> heights = **new** ArrayList<>();

heights.add(**null**);

**int** h = heights.get(0); *// NullPointerException*

* Otro caso interesante es en el método remove(), como vimos líneas arriba tiene dos formas de ser llamado: indicando la posición o el objeto en sí. En el ejemplo siguiente el resultado a pintar será 1 ya que en el remove() al pasar el literal estás ejecutando el método que elimina de acuerdo a la posición. Si deseas forzar a utilizar el objeto se tendría que hacer a la antigüita (numbers.remove(new Integer(1)).

List<Integer> numbers = **new** ArrayList<>();

numbers.add(1);

numbers.add(2);

numbers.remove(1);

System.out.println(numbers);

### Convirtiendo ArrayList y Array

* Vamos a empezar convirtiendo un ArrayList en un Array. En el ejemplo debajo vemos que lo más conocido está en la línea 6, lo malo en este caso es que Java convierte de forma predeterminada a un Object[] que no es lo que se desea por lo general. Ya en la línea 8 observamos aquello que es lo que se necesita, le indicamos el tipo de Array que se necesita (String[]). Otro punto importante es el tamaño del array, al colocar cero le indicamos a Java que el tamaño va a ser el mismo que tiene el ArrayList.

|  |  |
| --- | --- |
| 3  4  5  6  7  8  9 | List<String> list = **new** ArrayList<>();  list.add("hawk");  list.add("robin");  Object[] objectArray = list.toArray();  System.out.println(objectArray.length); *// 2*  String[] stringArray = list.toArray(**new** String[0]);  System.out.println(stringArray.length); *// 2* |

* Convertir de un Array a un ArrayList es más interesante ya que lo que realmente se hace es convertir el Array a una lista de tamaño fijo y esta lista con el array estarán conectados, cualquier cambio que se realice en uno se reflejará en el otro. Como se puede observar en el fragmento de código debajo, mientras el tamaño de la lista no se altere todo irá bien, por ello en la línea 26 arroja error al intentar eliminar el contenido de la posición 1.

20: String[] array = { "hawk", "robin" }; *// [hawk, robin]*

21: List<String> list = Arrays.asList(array); *// returns fixed size list*

22: System.out.println(list.size()); *// 2*

23: list.set(1, "test"); *// [hawk, test]*

24: array[0] = "new"; *// [new, test]*

25: **for** (String b : array) System.out.print(b + " "); *// new test*

26: list.remove(1); *// throws UnsupportedOperation Exception*

### Ordenamiento

* Es muy similar a ordenar un Array. Solo que a diferencia de los arrays no utiliza Arrays.sort sino Collections.sort.

List<Integer> numbers = **new** ArrayList<>();

numbers.add(99);

numbers.add(5);

numbers.add(81);

Collections.sort(numbers);

System.out.println(numbers); [5, 81, 99]

## Trabajando con Fechas y Horas

* A partir de Java 8, Oracle cambió completamente la forma de trabajar con fechas, en el examen no se encontrarán preguntas con la antigua forma, solo de la nueva API que se puede utilizar importando java.time.
* Lo primero a decidir al trabajar con fechas es cuanta información necesitamos:
  + LocalDate: Solo fechas
  + LocalTime: Solo la hora actual hasta nanosegundos
  + LocalDateTime: Contiene ambos, la fecha y la hora.
* Podemos obtener la fecha actual mediante la función now(), puede ser llamado con cualquiera de las 3 clases anteriores.
* Para inicializar las fechas tenemos dos caminos:

**public** **static** LocalDate of(**int** year, **int** month, **int** dayOfMonth)

**public** **static** LocalDate of(**int** year, Month month, **int** dayOfMonth)

LocalDate date2 = LocalDate.of(2015, 1, 20);

LocalDate date1 = LocalDate.of(2015, Month.JANUARY, 20);

* Por lo general los conteos de Java inician con cero pero en el caso de meses se inicia con 1.
* Para el tiempo podemos escoger cuan específicos deseamos ser:

**public** **static** LocalTime of(**int** hour, **int** minute)

**public** **static** LocalTime of(**int** hour, **int** minute, **int** second)

**public** **static** LocalTime of(**int** hour, **int** minute, **int** second, **int** nanos)

LocalTime time1 = LocalTime.of(6, 15); *// horas y minutos*

LocalTime time2 = LocalTime.of(6, 15, 30); *// + segundos*

LocalTime time3 = LocalTime.of(6, 15, 30, 200); *// + nanosegundos*

* Finalmente podemos combinar la fecha con el tiempo:

**public** **static** LocalDateTime of(**int** year, **int** month, **int** dayOfMonth, **int** hour, **int** minute)

**public** **static** LocalDateTime of(**int** year, **int** month, **int** dayOfMonth, **int** hour, **int** minute, **int** second)

**public** **static** LocalDateTime of(**int** year, **int** month, **int** dayOfMonth, **int** hour, **int** minute, **int** second, **int** nanos)

**public** **static** LocalDateTime of(**int** year, Month month, **int** dayOfMonth, **int** hour, **int** minute)

**public** **static** LocalDateTime of(**int** year, Month month, **int** dayOfMonth, **int** hour, **int** minute, **int** second)

**public** **static** LocalDateTime of(**int** year, Month month, **int** dayOfMonth, **int** hour, **int** minute, **int** second, **int** nanos)

**public** **static** LocalDateTime of(LocalDate date, LocalTime)

* Como puede observarse hay muchas formas de instanciar una fecha, hora y ambos, incluso se puede instanciar un LocalDateTime a partir de un LocalDate y un LocalTime.
* También hay que tener en cuenta que en ningún momento se está usando un constructor, las 3 clases tienen constructores privados para forzar el uso de los métodos estáticos.

LocalDate d = **new** LocalDate(); *// NO COMPILA*

* También se arrojará error si se intenta crear una fecha con números inválidos, por ejemplo, colocar como día un 32.

### Manipulando Fechas y Horas

* Añadir tiempo a una fecha es sencillo. La fecha y hora son **inmutables**, es decir que no pueden modificarse una vez creadas ya que no tiene setters, por ello al añadir o quitar meses, años, días, etc, no olvidar asignarlo a una variable de referencia (o a la misma).

|  |  |  |  |
| --- | --- | --- | --- |
| **METODOS** | **LOCALDATE** | **LOCALTIME** | **LOCALDATETIME** |
| plusYears/minusYears | Yes | No | Yes |
| plusMonths/minusMonths | Yes | No | Yes |
| plusWeeks/minusWeeks | Yes | No | Yes |
| plusDays/minusDays | Yes | No | Yes |
| plusHours/minusHours | No | Yes | Yes |
| plusMinutes/minusMinutes | No | Yes | Yes |
| plusSeconds/minusSeconds | No | Yes | Yes |
| plusNanos/minusNanos | No | Yes | Yes |

* Como se observa en la tabla anterior pueden añadirse distintos factores de tiempo, Java es suficientemente inteligente para distinguir años bisiestos y para mostrar segundos y nanosegundos cuando realmente se estén utilizando (cuando son ceros no). Tener cuidado con las preguntas con truco, por ejemplo:

LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);

date.plusDays(10);

System.out.println(date); *// MUESTRA LA FECHA ORIGINAL (INMUTABLE)*

LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);

date = date.plusMinutes(1); *// NO COMPILA*

* LocalDate y LocalDateTime tienen la posibilidad de convertirse a long (basados en el inicio 1970 cuando se comenzó a utilizar UNIX para el estándar de fechas):
  + LocalDate tiene toEpochDay(), el cual es el número de días desde 1970/01/01.
  + LocalDateTime tiene toEpochTime(), el cual es el número de segundos desde 1970/01/01.
* Para añadir periodos de tiempo tenemos la clase **Period**, la cual permite crear periodos de años, meses, semanas, días, y una combinación de años, meses y días.

Period annually = Period.ofYears(1); *// cada 1 año*

Period quarterly = Period.ofMonths(3); *// cada 3 meses*

Period everyThreeWeeks = Period.ofWeeks(3); *// cada 3 semanas*

Period everyOtherDay = Period.ofDays(2); *// cada 2 días*

Period everyYearAndAWeek = Period.of(1, 0, 7); *// cada año y 7 días*

* Un ejemplo del uso de **Period** es el siguiente; como podemos ver el periodo puede ser definido de diferentes formas, teniendo un posible método **performAnimalEnrichment()** reutilizable si se desea cambiar de periodo en futuras implementaciones:

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

LocalDate start = LocalDate.of(2015, Month.JANUARY, 1);

LocalDate end = LocalDate.of(2015, Month.MARCH, 30);

Period period = Period.ofMonths(1); *// crea un periodo*

performAnimalEnrichment(start, end, period);

}

**private** **static** **void** performAnimalEnrichment(LocalDate start, LocalDate end,

Period period) { *// uso del period genérico*

LocalDate upTo = start;

**while** (upTo.isBefore(end)) {

System.out.println("give new toy: " + upTo);

upTo = upTo.plus(period); *// se añade al periodo*

}

}

* Tener cuidado al definir un periodo, no puede encadenarse la llamada:

Period wrong = Period.ofYears(1).ofWeeks(1); *// cada semana*

Period wrong = Period.ofYears(1);

wrong = Period.ofWeeks(1); *// igual cada semana*

### Formateando Fechas y Horas

* Las clases vistas para fechas y horas cuentan con varios métodos para obtener la data:

System.out.println(date.getDayOfWeek()); *// MONDAY*

System.out.println(date.getMonth()); *// JANUARY*

System.out.println(date.getYear()); *// 2020*

System.out.println(date.getDayOfYear()); *// 20*

* Así como en versiones anteriores de Java también se cuenta con una clase de formateo de fechas: DateTimeFormatter, que en el ejemplo siguiente vemos el formateo con el estándar ISO de fechas.

LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);

LocalTime time = LocalTime.of(11, 12, 34);

LocalDateTime dateTime = LocalDateTime.of(date, time);

System.out.println(date.format(DateTimeFormatter.ISO\_LOCAL\_DATE));

System.out.println(time.format(DateTimeFormatter.ISO\_LOCAL\_TIME));

System.out.println(dateTime.format(DateTimeFormatter.ISO\_LOCAL\_DATE\_TIME));

2020-01-20

11:12:34

2020-01-20T11:12:34

* Ya que generalmente se desea un formato específico de fechas se tienen unos predeterminados de acuerdo a la Zona donde se encuentre.

DateTimeFormatter shortDateTime =

DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);

System.out.println(shortDateTime.format(dateTime)); *// 1/20/20*

System.out.println(shortDateTime.format(date)); *// 1/20/20*

System.out.println(shortDateTime.format(time)); *// UnsupportedTemporalTypeException*

DateTimeFormatter shortDateTime =

DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);

System.out.println(dateTime.format(shortDateTime));

System.out.println(date.format(shortDateTime));

System.out.println(time.format(shortDateTime));

* En el ejemplo debajo se muestran dos maneras de hacerlo para el formato corto de fechas (FormatStyle.SHORT), pero se debe tener cuidado ya que de acuerdo al tipo de formateador generado podría dar error y arrojar una excepción por tipo de fecha como se muestra en el ejemplo anterior. A continuación la tabla con las restricciones:

|  |  |  |  |
| --- | --- | --- | --- |
| **DateTimeFormatter f = DateTime Formatter.\_\_\_\_\_ (FormatStyle.SHORT);** | **Calling f.format** | **Calling f.format** | **Calling f.format** |
| ofLocalizedDate | Legal – shows whole object | NoLegal – shows just date part | Throws runtime exception |
| ofLocalizedDateTime | Throws runtime exception | Legal – shows whole object | Throws runtime exception |
| ofLocalizedTime | Throws runtime exception | Legal – shows just time part | Legal – shows whole object |

* Existen dos formatos predefinidos para mostrar fechas: FormatStyle.SHORT y FormatStyle.MEDIUM, existen otros pero envuelven zonas horarios que no vienen en el examen.

System.out.println(shortF.format(dateTime)); *// 1/20/20 11:12 AM*

System.out.println(mediumF.format(dateTime)); *// Jan 20, 2020 11:12:34 AM*

* Si no se desean utilizar los formatos predefinidos puede especificarse el formato deseado con un pattern.

DateTimeFormatter f = DateTimeFormatter.ofPattern("MMMM dd, yyyy, hh:mm");

System.out.println(dateTime.format(f)); *// January 20, 2020, 11:12*

* Un pequeño repaso, en el siguiente ejemplo el error se produce en la línea 6 debido a que un LocalDate no tiene horas incluidas.

4: DateTimeFormatter f = DateTimeFormatter.ofPattern("hh:mm");

5: f.format(dateTime);

6: f.format(date);

7: f.format(time);

* Para convertir String en Fechas se realiza de forma similar.

DateTimeFormatter f = DateTimeFormatter.ofPattern("MM dd yyyy");

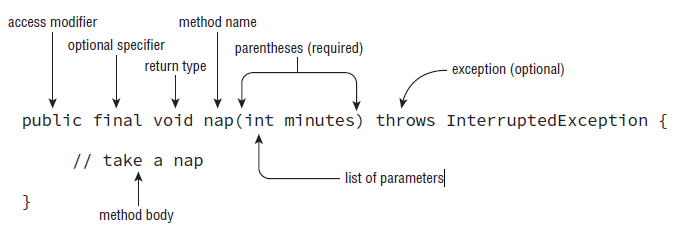
LocalDate date = LocalDate.parse("01 02 2015", f);

LocalTime time = LocalTime.parse("11:22");

System.out.println(date); *// 2015-01-02*

System.out.println(time); *// 11:22*

# Capítulo 4: Métodos y Encapsulamiento



## Diseñando métodos

* Java nos ofrece diversos modificadores de accesos para los métodos:
  + Public: el método puede llamarse de cualquier clase.
  + Private: el método solo puede llamarse dentro de la misma clase.
  + Protected: el método puede llamarse por clases del mismo paquete o por herencia (clases hijas pueden llamar a métodos de la clase padre aunque este en otro paquete)
  + Default: no se coloca la palabra default como modificador sino que no se coloca nada, en este caso el método puede llamarse por clases dentro del mismo paquete.
* Existen modificadores opcionales que pueden o no pueden ir, también que pueden ir juntos en el mismo método pero no todos son combinables: static (en este capítulo), abstract, final (siguiente capítulo), synchronized, native y strictpf (OCP).

**public** **void** walk1() {}

**public** **final** **void** walk2() {}

**public** **static** **final** **void** walk3() {}

**public** **final** **static** **void** walk4() {}

**public** modifier **void** walk5() {} *// NO COMPILA*

**public** **void** **final** walk6() {} *// NO COMPILA*

**final** **public** **void** walk7() {}

* Los tipos de retorno puede ser cualquier primitivo o un objeto, si no se desea retornar nada se utiliza **void,** pero siempre que se indica un tipo de retorno se debe retornar algo del mismo tipo indicado con el modificador **return.**
* Si se utiliza **void** si se puede utilizar **return** dentro del método pero vacío que lo que hará es terminar el método.
* Al utilizar un tipo de retorno y existe una condicional **if** dentro que envuelve el retorno deben asegurarse de colocar un **else** también ya que siempre debe retornar algo aunque la condición no se cumpla.
* Los nombres de los métodos siguen los mismos estándares para variables en Java: solo puede contener letras, números, $ o \_, pero no puede iniciar con números.
* Los parámetros son opcionales, así como la llamada **throws** (pueden haber múltiples separados por comas).

## Trabajando con Varargs

* Un método puede usar parámetros **Varargs**, solo se puede tener uno por método y debe estar al último.

**public** **static** **void** walk(**int** start, **int**... nums) {

System.out.println(nums.length);

}

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

walk(1); *// 0*

walk(1, 2); *// 1*

walk(1, 2, 3); *// 2*

walk(1, **new** **int**[] {4, 5}); *// 2 (ya que se le pasó como array)*

walk(1, **null**); *// throws a NullPointerException*

}

* No se puede enviar **null** al parámetro **Vararg** porque, de acuerdo al ejemplo, al no encontrar un **int** Java lo toma como si fuese un **array**, así que intenta obtener el tamaño de ese **array**, pero al ser **null** cae con error.

## Aplicando Modificadores de Acceso

* Los modificadores de acceso en orden de restricción del más restrictivo al menos restrictivo:
  + **Private:** solo se accede en la misma clase.
  + **Default**: **private** y se accede a clases en el mismo paquete.
  + **Protected**: **default** y se accede desde las clases hijas.
  + **Public**: **protected** y se accede de clases en otros paquetes.
* Aquí el modificador que tiene algunos casos algo confusos es el **protected** y es el que revisaremos.

**package** pond.shore;

**public** **class** Bird {

**protected** String text = "floating"; *// protected access*

**protected** **void** floatInWater() { *// protected access*

System.out.println(text);

}

}

**package** pond.swan;

**import** pond.shore.Bird; *// in different package than Bird*

**public** **class** Swan **extends** Bird { *// but subclass of bird*

**public** **void** swim() {

floatInWater(); *// package access to superclass*

System.out.println(text); *// package access to superclass*

}

**public** **void** helpOtherSwanSwim() {

Swan other = **new** Swan();

other.floatInWater(); *// package access to superclass*

System.out.println(other.text);*// package access to superclass*

}

**public** **void** helpOtherBirdSwim() {

Bird other = **new** Bird();

other.floatInWater(); *// DOES NOT COMPILE*

System.out.println(other.text); *// DOES NOT COMPILE*

}

}

* En el código anterior observamos que en el método **swim** tenemos acceso a los elementos **protected** de la clase **Bird** por la herencia de la clase **Swan.** En el método **helpOtherSwanSwim** también tenemos acceso ya que se instancia un objeto de **Swan** y se accede desde él mismo. En el método **helpOtherBirdSwim** no compila porque **Bird** no es subclase de **Bird** además que **Bird** se encuentra en otro paquete, y estamos utilizando una variable de referencia **Bird** no **Swan** (hija); tomar nota que la variable de referencia manda ya que es en tiempo de compilación que se detecta el error.

**package** pond.goose;

**import** pond.shore.Bird;

**public** **class** Goose **extends** Bird {

**public** **void** helpGooseSwim() {

Goose other = **new** Goose();

other.floatInWater();

System.out.println(other.text);

}

**public** **void** helpOtherGooseSwim() {

Bird other = **new** Goose();

other.floatInWater(); *// DOES NOT COMPILE*

System.out.println(other.text); *// DOES NOT COMPILE*

}

}

* Ahora en este nuevo fragmento de código vemos que en el segundo caso no compila, esto corrobora lo visto en el código anterior, pese a que el objeto será **Goose**, la variable de referencia es **Bird**, y **Bird** no es subclase de **Bird** y tampoco se encuentra en el mismo paquete en el que nos encontramos (variable de referencia manda).

**package** pond.duck;

**import** pond.goose.Goose;

**public** **class** GooseWatcher {

**public** **void** watch() {

Goose goose = **new** Goose();

goose.floatInWater(); *// DOES NOT COMPILE*

}

}

* En este último ejemplo vemos que pese a que ahora si estamos utilizando como variable de referencia a Goose y Goose es subclase de Bird no compila, esto debido a que no nos encontramos dentro de la clase Goose o en el mismo paquete de Bird.
* En resumen solo se tiene acceso a un protected estando en el mismo paquete o a través de la herencia pero estando dentro de la misma clase heredada y utilizando como variable de referencia a esta clase heredada si no se hace la llamada directamente (ya que nos encontramos en una subclase, podríamos llamar al método o variable protected directamente).
* Los métodos y variables estáticas son utilizadas para tener valores en común a todas las instancias (estado) de la clase y que no se necesita un objeto para obtenerlo, puede obtenerse directamente de la clase. Todas las instancias comparten ese estado y si se tiene métodos que solo usen variables estáticas, estos métodos deben también ser estáticos.
* De acuerdo a lo mencionado anteriormente, para llamar a una variable o método estático se realiza a partir del nombre de la clase, pero también podría realizarse con un objeto, con lo que tenemos un caso algo truculento:

**public** **class** Koala {

**public** **static** **int** count = 0; *// variable static*

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

System.out.println(count);

}

}

Koala k = **new** Koala();

System.out.println(k.count); *// k es un Koala*

k = **null**;

System.out.println(k.count); *// k es todavía un Koala*

* Java al llamar a una variable o método estático desde un objeto utiliza el tipo declarado, por ello, de acuerdo al ejemplo anterior, le da igual que ahora la variable de referencia apunte a **null**. Por ello recordar el tipo de la variable de referencia siempre que veas **static.**
* Un caso más ya que este tema es importante. Las variables estáticas se modifican para todas las instancias, por ello en el ejemplo debajo al final se pintará 5.

Koala.count = 4;

Koala koala1 = **new** Koala();

Koala koala2 = **new** Koala();

koala1.count = 6;

koala2.count = 5;

System.out.println(Koala.count); *//5*

* Las llamadas a métodos o variables estáticas tiene ciertas particularidades, por ejemplo un método **static** no puede llamar a una variable de referencia no **static**. El siguiente cuadrito lo explica.

|  |  |  |  |
| --- | --- | --- | --- |
| **Tipo** | **Llamada** | **Legal?** | **Cómo** |
| Método **static** | Otros método o variable **static** | Si | Usando nombre de clase. |
| Método **static** | Método o variable de instancia | No |  |
| Método de instancia | Método o variable **static.** | Si | Usando nombre de clase o variable de referencia |
| Método de instancia | Otro método o variable de instancia | Si | Usando variable de referencia. |

* Cuando se añade **final** a una variable **static** se está hablando de una constante y en Java se escribe el nombre con mayúsculas y un guion bajo entre palabras.
* Una variable **final** no puede reasignarse a otro objeto luego de su inicialización, lo que si puede realizarse es ejecutar métodos de la clase, por ejemplo un **ArrayList** **final**, una vez que ya se inicializó no puede reasignársele otra instancia pero si se puede añadir elementos con el método **add**.
* Recordar siempre que cuando se declara una constante (static final) debe inicializarse si o si, ya sea en la misma línea de declaración o en el bloque static.

**public** **class** Static {

**private** **static** **int** one;

**private** **static** **final** **int** two;

**private** **static** **final** **int** three = 3;

**private** **static** **final** **int** four; *// NO COMPILA PORQUE DEBE INICIALIZARSE*

**static** {

one = 1;

two = 2;

three = 3; *// NO COMPILA PORQUE YA SE INICIALIZÓ AL DECLARAR.*

two = 4; *// NO COMPILA PORQUE YA SE INICIALIZÓ EN EL BLOQUE MAS ARRIBA.*

}

}

* **Static** no solo se utiliza para variables y métodos, también se pueden realizar **static imports**. Los static imports se utilizan para importar miembros static de una clase (variables o métodos).

**import** static java.util.Arrays; *// NO COMPILA*

**import** static java.util.Arrays.asList;

**static** **import** java.util.Arrays.\*; *// NO COMPILA*

**public** **class** BadStaticImports {

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

Arrays.asList("one"); *// NO COMPILA*

}

}

* Java es un lenguaje pass-by-value, es decir que a los métodos se les pasa como parámetro una copia de la variable. Si es un primitivo es un valor simple, pero si es una variable de referencia se pasa una copia de esa variable pero que apunta al mismo objeto, por ello si dentro del método se le asigna a la variable otro valor u objeto, no afectaría a la variable inicial que se le pasó al método; otro asunto es si se llama a un método de la variable de referencia en el método ejecutado, ahí se está actualizando el objeto en sí, por ello ya fuera del método se verían cambios en la variable de referencia original (ejemplo debajo).

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

StringBuilder name = **new** StringBuilder();

speak(name);

System.out.println(name); *// Webby*

}

**public** **static** **void** speak(StringBuilder s) {

s.append("Webby");

}

## Métodos Sobrecargados (Overloaded)

* Los métodos sobrecargados son aquellos que tienen el mismo nombre pero diferente tipo o cantidad de parámetros, pueden además tener el mismo o diferente tipo de retorno así como algunos modificadores adicionales (static, etc) pero ello no influirá en que sea sobrecargado o no, son los parámetros quienes lo definen.