develop
talent & technology

GUÍA DE CERTIFICACIÓN

Fundamentos de Programación en Java

Fundamentos de Programación en Java | Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

AVISO LEGAL

Todos los derechos reservados. Queda estrictamente prohibido bajo las sanciones establecidas en las leyes, la reproducción parcial o total de esta obra por cualquier medio o procedimiento de alquiler o préstamos públicos de ejemplares de la misma, sin la autorización escrita de Develop Talent & Technology – 7i Business Solutions S.A. de C.V.

develop
talent & technology

INDICE

INDICE **3**

CONCEPTOS BÁSICOS DE ORIENTACIÓN A OBJETOS **5**

1.1 DEFINICIÓN DE CLASES Y OBJETOS 6

1.2 ESTRUCTURA DE UNA CLASE 6

1.2.1 Declaración de atributos *7*

1.2.2 Declaración de métodos *8*

1.2.3 Uso y declaración del método main *9*

1.3 COMPILACIÓN Y EJECUCIÓN DE UN PROGRAMA 9

1.4 USO DE UNA CLASE TEST 10

1.5 ESTRUCTURA DE UN ARCHIVO FUENTE. 12

TIPOS DE DATOS PRIMITIVOS Y OPERADORES **17**

2.1 TIPOS DE DATOS PRIMITIVOS 18

2.1.1 Tipos enteros *18*

2.1.2 Tipos punto flotante *19*

2.1.3 Tipos textuales *20*

2.1.4 Tipo booleano *21*

2.2 OPERADORES ARITMÉTICOS 21

2.3 PRECEDENCIA DE OPERADORES. 23

2.4 PROMOCIÓN Y CASTING 25

2.5 ÁMBITO DE LAS VARIABLES. 27

CREACIÓN Y USO DE OBJETOS **29**

3.1 VARIABLES DE REFERENCIA A OBJETOS 30

3.1.1 Declaración de variables de referencia *30*

3.1.2 Instanciación de objetos *31*

3.1.3 Inicialización de variables de referencia *32*

3.2 CÓMO SE ALMACENAN EN MEMORIA LAS VARIABLES DE REFERENCIA 33

3.3 MANEJO DE OBJETOS Y VARIABLES DE REFERENCIA 36

3.3.1 Garbage Collector *37*

OPERADORES Y CONSTRUCCIONES DE TOMA DE DECISIÓN **40**

4.1 USO DE OPERADORES RELACIONALES Y CONDICIONALES 41

4.1.1 Operadores relacionales *41*

4.1.2 Operadores condicionales *42*

4.2 SENTENCIA *DE TOMA DE DECISIÓN* 44

4.2.1 Sentencia if *44*

4.2.2 Sentencia if-else *44*

4.2.3 Sentencia if-else-if *45*

4.3 SENTENCIA *SWITCH* 45

4.3.1 Cláusula break. *46*

 Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

4.3.2 Cláusula default. *47*

MANEJO DE CICLOS **48**

5.1 Creación de ciclos *while* 49
5.2 Creación de ciclos *for* 49
5.3 Creación de ciclos *do-while* 50
5.4 Comparación de los ciclos 51
5.5 Ciclos anidados 51

ARREGLOS **53**

6.1 Arreglos unidimensionales 54
6.1.1 Arreglo de primitivos. *54*
6.1.2 Arreglo de objetos *56*
6.2 Acceso a valores de un arreglo. 56
6.3 Uso del ciclo *for* mejorado. 57
6.4 Uso de argumentos del método *main.* 58
6.5 Arreglos bidimensionales 59

MÉTODOS Y CONSTRUCTORES **62**

7.1 Sintaxis y estructura de los métodos 63
7.2 Invocación de métodos de la misma clase 64
7.3 Invocación de métodos de diferente clase 65
7.4 Uso de argumentos 66
7.4.1 Declaración e invocación de métodos con argumentos *67*
7.5 Declaración de métodos con valor de retorno 68
7.6 Sobrecarga de métodos 70
7.7 Modificadores de acceso 73
7.8 Uso del encapsulamiento 74
7.8.1 Métodos get y set 75
7.9 Creación de constructores 78
7.9.1 El constructor default 78
7.9.2 Sobrecarga de constructores 79

APÉNDICE **82**

Instalación y Configuración del JDK 83

Fundamentos de Programación en Java | Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

develop
talent & technology

MÓDULO 01

Conceptos básicos de orientación a objetos

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

1.1 Definición de clases y objetos

Para entender los términos **clase** y **objeto** se tendrá que hablar de dos paradigmas: el primero llamado paradigma estructural, el cual tiene como principal objetivo dividir un problema complejo en varios problemas menos complejos utilizando funciones. Sin embargo, con el tiempo el paradigma estructural ha quedado en desuso dando paso al paradigma Orientado a Objetos, el cual consiste en que el problema complejo se divida en elementos llamados objetos; dichos objetos son representaciones de lo que se conoce como entidades aplicado a la vida real.

El paradigma Orientado a Objetos a diferencia del estructural sólo se basa en acciones que le conciernen a determinados objetos, cada una de estas entidades es responsable de las acciones que le corresponden, por ejemplo, en un sistema bancario un gerente no debe realizar las acciones de un cuenta-habiente.

Primeramente es necesario definir uno de los conceptos más importantes de la orientación a objetos en Java: las clases. Una clase es aquella que construye una entidad, es otras palabras, una clase es quien determina las actividades (en Java son conocidas como métodos) y características (también llamadas atributos o variables de instancia) que le corresponden a un objeto. Continuando con el ejemplo del sistema de un banco, una clase sería el Gerente y otra sería Banco, cada uno de los objetos que sean creados a partir de ellas sólo podrán realizar actividades y poseer características propias de un Gerente o de un Banco respectivamente.

Por definición, todos los objetos en Java son creados a partir de un prototipo que determina cuáles serán sus características y comportamientos, este prototipo es la clase. Una clase es una especie de plano maestro para crear objetos de un mismo tipo; por ejemplo, al construir varios automóviles del mismo modelo existe un plano o diagrama que contiene todas las especificaciones que deben llevar dichos automóviles para poderlos fabricar o, según la terminología de Java, poderlos instanciar.

Existen otros criterios para definir los objetos, por ejemplo, saber si se trata de entidades físicas o conceptuales. Los objetos físicos son aquellos que representan cualquier cosa tangible en la realidad, como una botella de agua, un lápiz, una mesa, etcétera; por otra parte, los objetos conceptuales no necesariamente deben existir en la realidad o representarse de forma material, por lo que se debe realizar un análisis detallado para abstraer sus características para determinar si efectivamente se trata de un objeto; por ejemplo, la temperatura podría definirse como un atributo de un objeto, sin embargo, también es posible adjudicarle la categoría de objeto puesto que puede tener sus propias características (escala, magnitud) y realizar operaciones (cambiar escala, subir o bajar de magnitud).

1.2 Estructura de una clase

La sintaxis para declarar una clase en Java está dada por contener un modificador de acceso que puede ser ***public*** o ***friendly-package*** (*default*), el primero provoca que la clase sea visible a todo el universo (todo el sistema), mientras que el segundo hace que la clase sólo sea visible por las clases que están dentro del mismo paquete, esto se explicará a detalle más adelante. Una vez definido el nivel de acceso sigue la palabra reservada ***class*** que denota que se utilizará una entidad; posteriormente el ***nombre*** que tendrá la clase, este nombre debe escribirse con la primera letra mayúscula, utiliza nomenclatura *PascalCase* y el nombre debe ser un sustantivo. Finalmente se indica un bloque de código dado por 2 llaves que indican el alcance de esa clase, dentro de éstas se definen atributos y métodos.

```
Sintaxis:
[modificadores] class NombreClase {
   //atributos, constructores y métodos
}

Ejemplo 1:
public class Banco {
   //atributos, constructores y métodos
}

Ejemplo 2:
class Gerente {
   //atributos, constructores y métodos
}
```

1.2.1 Declaración de atributos

Como se indicó anteriormente, los **atributos** son las características que le pertenecen sólo a las instancias (objetos) de una clase, de ahí su nombre **variables de instancia**. Para entender esto volvamos al ejemplo del Banco y sus Gerentes: se sabe hasta ahorita que Banco y Gerente serian clases y que además los objetos serían las instancias de esas clases; también se sabe que el Gerente puede tener la característica *nombre*, pero si se tienen 2 gerentes en el banco cada uno de ellos debería tener su propio nombre; por esa razón se dice que existen 2 instancias de Gerente y cada objeto Gerente posee su propio nombre o, dicho de forma correcta, cada uno tiene su propia variable de instancia *nombre*.

Para la declaración de una variable de instancia o atributo primero se define - dentro del bloque de la clase - uno de 4 posibles modificadores de acceso, los cuales pueden ser ***private***, ***friendly-package***, ***protected*** y ***public***, posteriormente se indica el tipo de dato que almacenará la variable; luego se define el nombre de la variable utilizando la nomenclatura camelCase, se recomienda utilizar nombres cortos y con significado explícito. Es opcional asignar un valor para la variable de instancia, si se opta por colocar un valor inicial se utiliza el operador "=" después del nombre y en seguida el valor que dependerá del tipo de dato utilizado.

```
Sintaxis:

   [modificadores] tipo nombreDeVariable [= valor];

Ejemplos:

class Gerente {
   private String nombre = "Salvador";
   public String email;
   public String apellidoPaterno;
   int edad = 25;
}
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

1.2.2 Declaración de métodos

Los métodos se definen dentro del bloque de la clase y como buena práctica se colocan después de la declaración de los atributos. A diferencia de estos últimos, los métodos tienen su propio bloque de código en donde se definen actividades que éste ejecutará, las cuales pueden utilizar los atributos de la clase; cabe mencionar que si declaran variables dentro del método, sólo se podrán utilizar dentro de ese contexto.

La sintaxis básica de un método es la siguiente: un modificador, por el momento sólo se mencionarán los 4 modificadores de acceso, ***private, firendly-package (default), protected o public***, posteriormente se indica el tipo de retorno que puede ser un primitivo o uno de tipo referencia a objeto o indicar que el método no retornará ningún valor mediante la palabra reservada ***void***. Enseguida va el nombre del método en donde la primer letra debe ser minúscula, utiliza nomenclatura camelCase y los nombres deben formarse por el par nombre verbo + sustantivo; luego un par de paréntesis que abran y cierren para indicar los argumentos (o mejor conocidos como datos de entrada) que pueden ser opcionales y separados por comas; finalmente un par de llaves que delimitan el alcance del método.

Como último punto a considerar, si en el tipo de retorno se especifica primitivo o de referencia deberá ser obligatoriamente utilizada la palabra reservada **return** como última sentencia de ejecución del método.

Sintaxis:

```
    [modificadores] tipoDeRetorno nombreDeMetodo ([argumentos]) {
       //cuerpo del método
    }
```

Ejemplo:

```
class Gerente {

   private String nombre = "Salvador";
   public String email;
   int edad = 25;

   public void mostrarNombre() {
      System.out.println("Nombre: " + nombre);
   }

   public int consultarEdad() {
      return edad;
   }
}
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

1.2.3 Uso y declaración del método main

El método principal (**main**) es un método especial donde inicia la ejecución del programa, el cual debe ser declarado dentro del bloque de la clase, si la clase no contiene método principal no podrá ejecutarse.

La sintaxis del método main consta del modificador de acceso **public**, del modificador **static** el cual le permite ejecutarse sin hacer una instancia de la clase que lo contenga, tipo de retorno **void**, el nombre del método debe ser estrictamente **main** en minúsculas (de lo contrario será otro método). Algo importante que se debe conocer es que Java es sensible de mayúsculas y minúsculas, y entre los parámetros o argumentos debe recibir un arreglo de cadenas, el nombre de las variables no necesariamente debe llamarse *args*.

Sintaxis:

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

Ejemplo:

```
public class TestBanco{

   public static void main(String[] parametros) {
      //código que va a ejecutar el método main
   }

}
```

1.3 Compilación y ejecución de un programa

Cuando se codifica en java una clase, se almacena en un archivo con extensión **.java**, el cual se conoce como archivo fuente, este documento se debe compilar para producir un archivo con extensión .**class** que es código byte (lenguaje máquina), este archivo lo interpreta la Máquina Virtual de Java (JVM), encargada de solicitar los recursos al Sistema Operativo.

Algo importante que se debe conocer es que un archivo fuente puede contener la declaración de varias clases pero solo puede haber una clase pública por cada archivo fuente, si existe una clase pública, el nombre de la clase y el nombre del archivo fuente deberán llamarse igual, además, un archivo fuente puede tener la declaración de varias clases pero hay que recordar que sólo una podrá ser *public* y las demás deberán conservar el nivel de acceso friendly-package (sin modificador). Las clases que tienen modificador por default pueden contener cualquier nombre y éstos no deben llamarse igual que el archivo.

Para la creación de un archivo java, es necesario codificar la clase en un archivo fuente, se recomienda como una excelente práctica contener una clase por archivo fuente. Para realizar esta sencilla tarea de compilación se recomienda utilizar los siguientes pasos:

1. Una vez que se generó el “Archivo.java” se almacena en algún directorio.
2. Desde la consola de comandos (CMD o Terminal), posicionar el *promt* sobre el directorio donde se almacena el archivo fuente.
3. Antes de utilizar la instrucción ***javac*** se debió configurar la variable de entorno y tener instalado el JDK, en dado caso de no ser así véase Apendice A , una vez configurado se utiliza la siguiente instrucción.

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

```
C:\DirectorioProyecto>javac NombreArchivoFuente.java
```

4. Si todo marcho bien, se debe regresar al mismo Prompt sin ningún mensaje, de lo contrario se deben corregir los errores de compilación.

Al realizar los pasos anteriores se debió crear uno o varios archivos con la extensión .class y con el nombre de la clase o nombres de las clases que tiene el archivo fuente.

Una vez realizado esto se debe hacer el siguiente paso utilizando la instrucción ***java***, el cual es el encargado de crear una instancia de la JVM sobre la cual correr la aplicación, con la cual se vera la ejecución del programa.

Para la ejecución de una clase cabe mencionar que solo podrán ejecutar las clases que contengan la firma adecuada del método principal de lo contrario no podrá ejecutarse y el error será en tiempo de ejecución conocidos mejor como *exceptions* **NotSuchMethodMain**, se pueden realizar la siguiente forma.

1. Navegar hasta el directorio donde se encuentra el archivo .class.
2. Ejecutar el comando java con el Nombre de la clase sin extensión como se muestra en la figura.

```
C:\ DirectorioProyecto >java NombreClase
```

Si se encuentra en algún otro directorio que no sea el proyecto donde se almacena el archivo .class podemos utilizar una opción de buscador de clases especifico de java, la cual es llamada classpath cuyo principal objetivo es localizar los recursos con la siguiente instrucción, **java** seguido de la opción de la máquina virtual **–classpath** o utilizarlo de forma corta **–cp**, seguido del directorio donde se encuentra el .class y finalizar con el nombre de la clase.

```
C:\ OtroDirectorio >  java -classpath C:\DirectorioProyecto NombreClase
```

```
C:\ OtroDirectorio >  java -cp C:\DirectorioProyecto NombreClase
```

1.4 Uso de una clase test

El objetivo de las clases con el prefijo Test es probar nuestros modelos o demás clases con el fin de detectar inconsistencias, probar métodos, comprobar valores e interactuar con otros objetos mientras relaciones.

Para llevar a cabo este ejercicio, se tomará el ejemplo del banco y sus gerente, para empezar se recomienda como buena práctica empezar a codificar aquellas clases independientes, es decir primero modelos y posteriormente las clases de prueba.

```
//Archivo nombrado Gerente.java

class Gerente {
   private String nombre = "Salvador";
   public String email;
   int edad = 25;

   public void mostrarNombre() {
      System.out.println("Nombre: " + nombre);
   }

   public int consultarEdad() {
      return edad;
   }
}

   public static void main(String[] parametros) {
      //código que va a ejecutar el método main
```

Como se mostró en la figura anterior se recomienda codificar una clase en un archivo llamado Gerente.java, enseguida se codifica la clase `TestGerente` en un archivo llamado TestGerente.java como el cual hará una instancia de la clase `Gerente` y probara algunos atributos y métodos, el atributo nombre no se podrá utilizar debido a que como tiene modificador ***private*** significa que solo es visible en la clase `Gerente`, por lo cual no podrá ser visto en la clase `TestBanco`, pero si podrá ser utilizado el método `mostrarNombre()`

```
//Archivo nombrado TestBanco.java

public class TestBanco{

   public static void main(String[] args) {
      Gerente gComercial= new Gerente();
      gComercial.email = "shernandez@7i.com.mx";
      gComercial. edad = 25;

      System.out.println("E-mail: " + gComercial.email);
      System.out.println("Edad: " + gComercial.consultarEdad());
      gComercial.mostrarNombre();
   }
```

La finalidad de la clase `TestBanco` es probar una instancia de gerente que opera el banco, una vez que ambos archivos fuentes se guarden en el mismo directorio, se compilara con la siguiente instrucción, el cual indica que se compilaran todos los archivos fuente que se encuentre en el directorio del proyecto, generando ambos archivos .class.

```
C:\DirectorioProyecto>javac *.java
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

Una vez compilado, el siguiente paso es ejecutar solo la clase que tiene el método principal (TestBanco) con la siguiente instrucción estando bajo el directorio del proyecto, como se muestra a continuación.

```
C:\ DirectorioProyecto >java TestBanco
```

El resultado de la ejecución debe ser muy similar a la siguiente:

```
E-mail: shernandez@7i.com.mx
Edad: 25
Nombre: Salvador
```

1.5 Estructura de un archivo fuente.

La estructura de un archivo fuente tanto como del proyecto tiene que ver con la manera de administrar toda la información, modelos, clases test, recursos, etc.; ya que con el ejemplo anterior en el mismo directorio se tienen modelos, clases de prueba y archivos .class, para ello es necesario las palabras reservadas **package** el cual permite indicarle al archivo fuente que se encuentra bajo un estructura de directorios que es parte del proyecto con el fin de tener un mejor control de clasificar contenidos como por ejemplos modelos y clases de prueba , esta palabra debe ser antes de la declaración de las clases; la otra palabra reservada es **import** la cual permite indicar que el archivo fuente hará una dependencia de otra clase o clases que se encuentra en una estructura de paquetes y esta debe ir como antes de la declaración de las clases y depués del indicador de **package**.

Cabe mencionar que solo puede existir una palabra **package** y varios **import** por archivo fuente.

La sintaxis para indicar que el archivo fuente se encuentra en un paquete es mediante la palabra reservada package seguido de las rutas donde queremos que quede alojada, separados por puntos para indicar un subdirectorio, se recomienda utilizar un identificador único para la aplicación y que mejor que utilizar el dominio de la empresa y también como buena práctica se recomienda utilizar el nombre en minúsculas cuya codificación queda de la siguiente manera.

Sintaxis:

```
package     path.subpath;
class       NombreClase{ }
```

La sintaxis de **importación** es muy parecida a la de empaquetar, la principal diferencia es que indicas qué clase es la que depende, ya que es parte del proyecto. La sintaxis queda de la siguiente manera, palabra reservada **import** seguido de los directorios donde se encuentra la clase o las clases a depender y finalmente el nombre de la clase o clases, si existen varias clases que se necesiten importar se puede optar por colocar "*" que significa dependencia de todas las clases que se encuentre bajo el esquema de paquetes.

```
Sintaxis:
      package     path.subpath;
      import      path.subpath.NombreClase;
      import      path.subpath.*;

      class       NombreClase{ }
```

Ahora bien, se tiene la siguiente estructura de archivos bajo el directorio del proyecto como se muestra en la siguiente figura aplicando el comando **tree** del sistema operativo Windows el cual muestra la estructura de directorios en forma de árbol del directorio actual indicado por el ".".

```
C:\ DirectorioProyecto >tree .
|____src
|
|____compilados
```

El propósito de realizar esto es que el directorio **src** contendrá los archivos fuente(.java) mientras que el directorio compilados los .class, para ellos se modifican los archivos fuente colocados en el directorio src.

```
//Gerente.java:

package mx.com.develop.banco.models;
class Gerente {

   private String nombre = "Salvador";
   public String email;
   int edad = 25;

   public void mostrarNombre() {
      System.out.println("Nombre: " + nombre);
   }

   public int consultarEdad() {
      return edad;
   }
}
```

La figura anterior nos indica que la clase Gerente pertenece a un paquete llamado mx seguido de sub paquetes llamados com, develop, banco y models.

A continuación la modificación de la clase TestBanco que nos indica también que se encuentra en un esquema de paquetes y que la clase TestBanco depende de la clase Gerente, por lo cual la modificación cambia de la siguiente manera.

```java
//TestBanco.java
package mx.com.develop.banco.tests;
import  mx.com.develop.banco.models.Gerente;

public class TestBanco{

   public static void main(String[] args) {
      Gerente gComercial= new Gerente();
      gComercial.email = "shernandez@7i.com.mx";
      gComercial. edad = 25;

      System.out.println("E-mail: " + gComercial.email);
      System.out.println("Edad: " + gComercial.consultarEdad());
      gComercial.mostrarNombre();
   }
```

Como se mencionó la clase TestBanco se encuentra bajo un esquema de paquetes llamado mx.com.develop.banco.tests, pero debido a que la clase TestBanco depende de la clase Gerente es necesario realizar una importación debido a que la clase se encuentra bajo un esquema de empaquetado y se encuentra en otro directorio con terminación models.

Una vez almacenados los archivos fuente en el directorio src de nuestro directorio del proyecto la estructura de archivos queda de la siguiente manera.

```
C:\ DirectorioProyecto >tree .
|____src
|      |_____ Gerente.java
|      |
|      |_____ TestGerente.java
|
|____compilados
```

El siguiente paso sería la generación de los archivos .class pero en el directorio *compilados* debido a que se necesita tener la separación de archivos fuente y archivos compilados, para ello se sabe que nuestro prompt se encuentra en el directorio de nuestro proyecto.

Por lo cual se puede optar por cambiar la sintaxis de compilación, debido a que el prompt no se encuentra el directorio src, ahora bien para resolver el problema de que los .class aparezcan en el directorio compilados es necesario colocar una opción después de indicar la herramienta javac y el nombre de la carpeta donde se requiere alojar lo .class quedando de la siguiente manera.

```
C:\DirectorioProyecto>javac –d compilados src/*.java
```

Esto indica que se compila todo aquello bajo el directorio src con extensión .java y con la opción –d indica que los archivos compilados (.class) se alojan en el directorio compilados bajo el directorio del proyecto quedando la estructura de archivos de la siguiente manera.

develop talent & technology

```
C:\ DirectorioProyecto >tree .
|____src
|      |____ Gerente.java
|      |
|      |____ TestGerente.java
|
|____compilados
       |____ mx
              |____ com
                     |____ develop
                              |____ models
                              |         |____ Gerente.class
                              |
                              |____ tests
                                        |____ TestGerente.class
```

Algo que se percata en la imagen anterior es que al indicarle que nuestro archivo fuente se encuentra con la palabra **package** directamente crea la estructura de archivos del paquete, ya que esos paquetes son parte del proyecto que sirven para administrar contenido, tanto `mx.com.develop.models y mx.com.develop.test` se crearan en el directorio compilados, no olvidar que esos paquetes forman parte del proyecto indicado por el archivo fuente y también no olvidar que tanto src y compilados no forman parte del proyecto puesto no se indicó en el archivo fuente .

La parte interesante de todo esto es la ejecución de la aplicación, se conoce hasta este momento que el **prompt** se encuentra en el directorio del proyecto y que los directorios **src** y **compilados** no se indica en ningún lado del archivo fuente que son parte del empaquetado.

Bien para la ejecución de la clase que se encuentra indicada por un paquete es necesario cambiar la sintaxis de la siguiente manera siempre y cuando los paquetes se encuentren enseguida en el mismo directorio.

```
C:\Directorio>java pack.subpack.Clase
```

Es decir que si nuestro prompt se localizara en el directorio ***compilados*** la ejecución quedaría de la siguiente manera.

```
C:\DirectorioProyecto\compilados > java mx.com.develop.tests.TestBanco
```

Pero sin embargo si se encuentra en algún otro directorio se necesitará indicar mediante un classpath donde encontrar esos .class con sus respectivos paquetes ya que fueron mencionados mediante la palabra reservada package.

Quedando de la siguiente manera.

```
C:\DirectorioProyecto > java –cp compilados mx.com.develop.tests.TestBanco
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

Como se ve en la figura anterior al encontrarse en el directorio del proyecto, y observar que compilados no forma parte del empaquetado pero este contiene los paquetes son sus respectivos .class se utiliza la opción –cp para buscarlos en ese directorio. Y posteriormente se indica el nombre de la clase con sus respectivos paquetes separados por puntos para la ejecución de la clase de prueba.

Quedando la salida de la siguiente manera:

```
E-mail: shernandez@7i.com.mx
Edad: 25
Nombre: Salvador
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

MÓDULO 02

Tipos de datos primitivos y operadores

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

2.1 Tipos de datos primitivos

Los tipos de datos de primitivos en Java son aquellos que guardan solo valores de acuerdo al tipo dato que se indique, son 8 los tipos de datos categorizados en 4 categorías que se describen a continuación.

2.1.1 Tipos enteros

Usados para representar un valor con parte entera, de los cuales en esta categoría pertenecen 4 tipos de datos primitivos representados por palabras reservadas, el límite de sus valores a contener están denotados por la cantidad de bits que soporta, la tabla siguiente muestra la cantidad de bits de cada tipo de dato y sus respetivos limites inferior y superior de valor posible a contener.

TIPO DE DATO	CANTIDAD DE BITS	LIMITE INFERIOR	LIMITE SUPERIOR
byte	8	-128	127
short	16	-32,768	32,767
int	32	-2,147,483,648	2,147,483,647
long	64	-1.8446744e+19	1.8446744e+19

La razón por la cual soporta esos límites es debido a que los valores son representados por la cantidad de bits que existen y como un bit puede ser entre 1 o 0, por ejemplo para el tipo de byte que consta de 8 bits se tiene en cuenta que solo ocupa 7 bits para conseguir el valor y el bit 8 es el encargado de dar el signo 0 para positivo y 1 para negativo.

Ejemplo:

El byte para conseguir el valor de límite superior se realiza la siguiente operación 2 ^ 7 = 128, pero considerar que el valor de 0 se toma como un valor positivo por lo cual será el límite superior de 127.

Mientras que para encontrar el límite inferior de un byte se realiza la operación -2 ^ 7 cuyo resultado es -128, y eso aplicara a los tipos de datos subsecuentes de tipo entero.

Para el caso de un tipo de dato long, cuando en el valor supere el límite superior de un entero se tendrá que especificar al finar del valor una “L” o una “l” de lo contrario marcara un error de compilación, debido a que el tipo de dato más utilizado en Java es un int el cual es tipo dato entero que está dado por default.

Ejemplos:

```
long valor1 = 2147483649;  //esto es un Error de compilación
long valor2 = 2147483649L; //Compila perfectamente
long valor2 = 214748; //Compila perfectamente
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

Ahora bien los tipos de datos enteros no solo soportan valores del sistema de decimal, también soporta el sistema octal y el sistema binario.

Para indicar un valor den sistema octal en el valor se antepone el **0** y se especifica un número del sistema octal(0 - 7).

```
int octal = 075;
```

Para indicar un valor del sistema hexadecimal en el valor se antepone **0x** y se especifica un conjunto de letras y/o números del sistema hexadecimal, ejemplo:

```
int hexadecimal = 0xcafe;
```

Para un valor binario solo a partir de la versión 7 en adelante se puede utilizar, en el valor anteponiendo un **0b** y posteriormente números del sistema binario(0 - 1), como se demuestra en la siguiente figura.

```
int binario = 0b1010;
```

De no contener un numero en el sistema octal, hexadecimal o binario marca un error de compilación como se muestra en los ejemplos.

```
int binario = 0b1020; //el 2 no corresponde al sistema binario
int hexadecimal = 0xcafz;     //la letra z no corresponde al sistema
                              //hexadecimal
int octal = 085; //el 8 no corresponde al sistema octal
```

Otra de las características que tiene que solo es usada a partir de la versión 7 de Java es que puede utilizar la separación de cifras cuando se traten de cifras muy grandes utilizando un "_" que no puede ir al inicio o al final del valor, solo entre los números, el cual permite tener una mayor legibilidad cuando se coloca un valor y resulta ser una mejor manera de leerlo más rápidamente, ejemplo:

```
int  monto = 2_147_000;
```

2.1.2 Tipos punto flotante

Esta categoría se diferencia por poder contener una parte entera y una parte decimal, existen 2 tipo de datos dados también por palabras reservadas, como se muestra a continuación.

TIPO DE DATO	CANTIDAD DE BITS
float	32
double	64

Aquí el dato por default en cuanto a tipos flotantes es el de 64 bits, sin embargo cuando se ocupa el tipo de dato **float** y este lleva una parte decimal en la cifra deberá llevar una "F" o una "f" de lo contario marcara un error de compilación.

Ejemplos:

```
      double valor1 = 45;                  //Compila perfectamente
      double valor2 = 45.67;         //Compila perfectamente
      float valor3 = 45;                   //Compila perfectamente
      float valor4 = 45.69F;         //Compila perfectamente
      float valor5 = 45.69;          //Error de compilación, es necesario
 especificar que se trata de un valor con punto flotante "f"
```

Otra de las características que tiene esta categoría es que puede especificarse mediante una notación científica con el fin de manejar exponentes y acortar la cifra.

Ejemplo:

```
      double valor1 = 45.67E7;                  //Compila perfectamente
      float valor1 = 45.67E7F;                  //Compila perfectamente
      //debe conservarse la "f" para que compile perfectamente.
```

2.1.3 Tipos textuales

En cuanto el tipo Textual solo existe 1, el cual es el **char,** con la capacidad de 16 bits y a diferencia de las categorías previas este es sin signo (unsigned), por lo cual se debe tomar en cuenta la siguiente tabla como referencia.

TIPO DE DATO	CANTIDAD DE BITS	LIMITE INFERIOR	LIMITE SUPERIOR
char	16	0	65535

Los valores que soporta el tipo de dato char, son un numero entero denotado dentro de los limites, como se muestra en el ejemplo, si se imprime el contenido de ese carácter lo tomara de la tabla UNICODE.

```
      char  oneChar = 64; // verdaderamente contiene el character  @
```

Otro valor que puede contener es solo un carácter, denotado por comillas simples, si se tiene más de un carácter marca un error de compilación.

```
      char  oneChar = '@';
```

El siguiente valor puede contener, una sentencia de escape, encerrado entre comillas simples y utilizando \ (backslash) seguido del carácter a escapar, se tiene en cuenta que tanto ***\t,\n,\r*** etc. Son caracteres que manipulan espaciado, aunque podemos escapar otros tipos de caracteres que no necesariamente son espaciados como se muestra en la siguiente figura.

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

```
char  oneChar = '\t'; // realmente  contiene una tabulacion
char  twoChar = '\\'; // realmente  contiene el caracter \
```

Finalmente el ultimo valor que puede contener es código Unicode, encerrado entre comillas simples seguido de un backslash (\ ) seguido de cuatro caracteres del sistema hexadecimal (no difiere el utilizar mayúsculas y minúsculas) y el valor que contendrá dependerá del carácter que se encuentre en la tabla Unicode.

```
char  oneChar = '\uCAFE'; // realmente  contiene el caracter �
```

2.1.4 Tipo booleano

A diferencia de otros lenguajes en donde se puede utilizar 0 y 1 para inicializar la variable, en Java solo es permitido las literales true y false (en minúsculas), o cualquier operación siempre y cuando regrese una expresión booleana.

Como se muestran en los ejemplos:

```
boolean  verdad = true  ;
booelan  resultado = (5 != 6);
```

2.2 Operadores aritméticos

En cuento a los operadores aritméticos, que permiten realizar operaciones y cabe destacar que se realizan de forma binaria, por ejemplo, si se tiene la expresión 7+5+9, no puede realizarse la operación de forma inmediatamente para ello es necesario realizar en forma de pares de operandos.

La siguiente tabla muestra los operadores que realizan operaciones matemáticas.

Operación	Operador	Ejemplo
Suma	+	a + b
Resta	-	a – b
Multiplicación	*	a * b
División	/	a / b
Módulo	%	a % b

Es importante mencionar que dependiendo la operación que se realiza es el tipo de dato que se tendrá al final, por ejemplo si se tiene un tipo de dato doublé OPERADOR int, al final se tendrá un resultado como parte doublé.

También importante indicar que en la división tanto como en la parte del módulo si se trabaja con tipos de datos de punto flotante se tendrá la parte flotante como resultado, y si es parte entera solo se contendrá la parte entera.

Ejemplos:

```
int   w = 5 ;
int   x = 9 ;
double  y = 6 ;
double  z = 4.0 ;
double  z1 = 9.0 ;

System.out.println( w + x); // 14 se imprime
System.out.println( x * y); // 54.0, ya que es un int * double=double
System.out.println( x + y); // 15.0, ya que es un int + double=double
System.out.println( x % y); // 3.0, un int / double y el
                              residuo de esa división es parte flotante.
System.out.println( y / z); // 1.5, debido a ambas variable son
                              fraccionarias.
System.out.println( w / x); // 0, debido al ser enteros no hará la
                              división con puntos flotantes
System.out.println( w / z1); // .5555555, debido al ser un operando
                               parte fraccionara todo el resultado será
                              fraccionario.
```

Existen 2 operadores que sirven para manipular incrementos o decrementos de una unidad al vez(sumar y restar) en dos versiones forma prefija y posfija, el cual es importante cuando se está realizando en medio de un cálculo o antes de terminar la sentencia, la forma de donde se encuentre el operando indica que hacer, por ejemplo la forma prefija indica que primero incrementa o decrementa y posteriormente utiliza el valor antes de pasar al siguiente evaluación, mientras que la forma posfija, primero utiliza el valor que tiene en ese momento y posteriormente incrementa o decrementa en una unidad.

Estos operadores se pueden aplicar tanto a tipos de datos enteros como de punto flotante y en la forma posfija se puede representar como 2 instrucciones separadas.

TIPO DE DATO	FORMA PREFIJA	FORMA POSFIJA	EQUIVALENCIA PREFIJA	EQUIVALENCIA POSFIJA
++	++OP	OP++	OP=OP+1;	OP; OP+1;
--	--OP	OP--	OP=OP-1;	OP; OP-1;

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

Algunos ejemplos de cómo se demuestra esta parte es:

```
int  w = 5 ;


System.out.println( w ); // 5, solo se imprime el contenido
System.out.println( w++ ); // 5, puesto primero lo imprime y
                           posteriormente se incrementa en una unidad.
System.out.println( w); // 6, al incrementarse previamente se
                         conserva el valor de 6.
System.out.println( ++w); // primero incrementa y posteriormente lo
                          utiliza imprimiendo el valor 7
System.out.println( w);// finalmente se conserva el valor de 7
```

2.3 Precedencia de operadores.

El orden de precedencia no es más que otra cosa que prioridad sobre otros operandos, por ejemplo si se tiene la siguiente operación 5 + 7 + 4 sería fácil encontrar el resultado, puesto todos son sumas, pero si la operación cambiara a 5 + 7 * 4, podríamos tener el valor de 48 o 33, pero la realidad es que debe ser 33.

La siguiente tabla resumida muestra el orden de precedencia así como también la evaluación si se lee de izquierda a derecha o viceversa.

Orden de Precedencia	Operador	Evaluación	Descripción
1	()	IZQ a DER	Paréntesis
2	++, --	IZQ a DER	Incrementos y decrementos, incluyendo forma prefija y posfija
3	*, / , %	IZQ a DER	Operaciones de multiplicación, división y residuo.
4	+, -	IZQ a DER	Sumas y restas.
5	<, >, >=, etc.	IZQ a DER	Operaciones de comparación.
6	&&, ||, etc.	IZQ a DER	Operaciones de condición.
7	=, +=, etc.	DER a IZQ	Operaciones de asignación.

Ejemplo si se requiere realizar la siguiente operación:

```
int  w = 5 ;

System.out.println( w++ + ++w +(5/2*4+23%3+ w++) - 78+ ++w - w % 2);
//1. Realizar los paréntesis.
                                 +(5/2*4+23%3+ w++)
//2. Se detecta que dentro de estos existe un post incremento, lo cual
indica que se utiliza como 5 pero para la siguiente vez que quiera utilizarse
tiene el valor 6.
                                 +(5/2*4+23%3+5)
```

Ahora, como se menciona la variable “w” será usada dentro del paréntesis como 5 pero la siguiente vez que se ocupe a w tendrá el valor de 6.

Terminando la operación de paréntesis queda de la siguiente forma:

```
int  w = 5 ;

System.out.println( w++ + ++w +(5/2*4+23%3+ w++) - 78+ ++w - w % 2);
//3. Recordar que se evalúa primero las * /  y % de izq. a der.
                        +(5/2*4+23%3+5)
                        +(2*4+23%3+5)// div de enteros, parte entera
                        +(8+23%3+5)
                        +(8+2+5)
                        +(10+5)
                        +(15)
//4. La siguiente operación quedaría así.
//Donde w= 6
//                  (w++ + ++w 15 - 78+ ++w - w % 2)
```

Continuando con la siguiente prioridad es incrementos y decrementos.

```
int w = 5 ;

System.out.println( w++ + ++w +(5/2*4+23%3+ w++) - 78+ ++w - w % 2);
//5. Realización de incrementos (de izq. a der)
//Donde “w” comienza con el valor de 6.
//                  (w++ + ++w + 15 - 78+ ++w - w % 2)
                    (6 + ++w + 15 - 78+ ++w - w % 2)
//Se utiliza como 6, pero al pasar al siguiente operando tendrá el //valor
de 7.  W = 7
                    (6 + 8 + 15 - 78+ ++w - w % 2)
//Como se pudo observar ahora es un 8, debido a que es un pre incremento, y
primero se incrementó a 8 y se utiliza como un 8.
```

Para la siguiente modificación ahora w que tiene el valor de 8,

```
int  w = 5 ;

System.out.println( w++ + ++w +(5/2*4+23%3+ w++) - 78+ ++w - w % 2);
//6. Donde w= 8
                    (6 + 8 + 15 - 78+ ++w - w % 2)
// se incrementa w a 9 y se utilizara como 9.
                    (6 + 8 + 15 - 78+ 9 - w % 2)
// la expresión quedaría de la siguiente manera:
                    (6 + 8 + 15 - 78+ 9 - 9 % 2)
```

Una vez que se ha reducido a sumas y restas, la operación queda de la siguiente manera:

```
int  w = 5 ;

System.out.println( w++ + ++w +(5/2*4+23%3+ w++) - 78+ ++w - w % 2);
                        (6 + 8 + 15 - 78+ 9 - 1)
                        (14 + 15 - 78+ 9 - 1)
                        (29 - 78+ 9 - 1)
                        (-49 + 9 - 1)
                        (-40 - 1)
                        (-41)
```

2.4 Promoción y casting.

La promoción se refiere a la capacidad de una variable de mayor capacidad en bits contener una variable de menor capacidad en bits de otra sin afectar el valor, esta promoción ocurre de forma automáticamente.

Por ejemplo, si recuerda en el caso de los tipos de datos primitivos el long es el que puede soportar cualquier tipo de dato inferior, la siguiente tabla mostraría la forma de promocionar los tipos de datos.

Tipo de dato	Permitido a contener
byte	Así mismo.
short	Así mismo + los anteriores.
char	Así mismo
int	Así mismo + los anteriores.
long	Así mismo + los anteriores.
float	Así mismo + los anteriores.
double	Así mismo + los anteriores.

Como ejemplo se coloca al float, el cual puede soportar la mayoría de los tipos de datos excepto el tipo de dato double.

La siguiente figura nos muestra formas validas, de aplicar promoción de forma correcta correspondiente a la tabla:

```
int  int32 = 5 ;
char char16= 'A';
long  long64 = 999 ;
//Aplicando promotion
float   float32 = int32;
        float32 = char16;
        float32 = long64;
```

Una forma de aplicar una promoción invalida seria de la siguiente manera cuando una de menor categoría querrá contener una de mayor categoría de acuerdo a la tabla, como se muestra a continuación.

```java
int  int32 = 5 ;
char char16= 'A';
long  long64 = 999 ;
//Aplicando promotion incorrectamente
short   short16 = int32;  //Error de compilacion.
        short16 = char16; //Error de compilacion.
        short16 = long64; //Error de compilacion.
```

Como se pudo apreciar en la figura anterior se muestra que marcan errores de compilación debido a que quiere contener datos que están por encima de su categoría, pero algo importante que se debe considerar , es que los valores de las variables en tiempo de ejecución si la pueden contener, pero en tiempo de compilación no se puede por ser menor rango, es por ello que es necesario realizar lo que se conoce como casting el cual es forzar un tipo de dato a contenerlo a otro, pero se tiene que tener precaución debido a que si supera el límite puede ocurrir un desbordamiento de valores o en su defecto puede ocurrir una pérdida de precisión como se muestra con los siguientes ejemplos, la sintaxis de la operación es después del igual colocar un paréntesis y forzarlo a un tipo de dato igual o inferior que el declarado al asignarlo.

```java
int  int32 = 5 ;
char char16= 'A';
long  long64 = 999 ;
//Aplicando casting
short   short16 = (short)int32;  //
        short16 = (byte)char16; //
        short16 = (short)long64; //
```

El siguiente ejemplo muestra una forma incorrecta de aplicar un casting.

```java
int  int32 = 5 ;
//Aplicando casting
short   short16 = (int)int32;  // debido a que un short de 16 bits contiene
una variable que a pesar del forzado indicado sigue siendo un int
```

La siguiente figura muestra una forma correcta de hacer casting pero la diferencia es que existe perdida de precisión y valor es truncado a 5, ya que solo se especifica que le interesa la parte entera

```java
float  int32 = 5.67f ;
//Aplicando casting
int   int32= (int)float32; // El valor que contiene int32 es  5.
```

Y finalmente cuando se aplica un casting también de forma valida, como por ejemplo cuando se declara un short con valor por arriba del límite superior de un byte, se tiene un desbordamiento de valor y dependiendo de cuanto se esté desbordando

para corregir esa anomalía, se tomara un bloque más del tipo de dato que fue forzado a comportarse empezado por los números negativos.

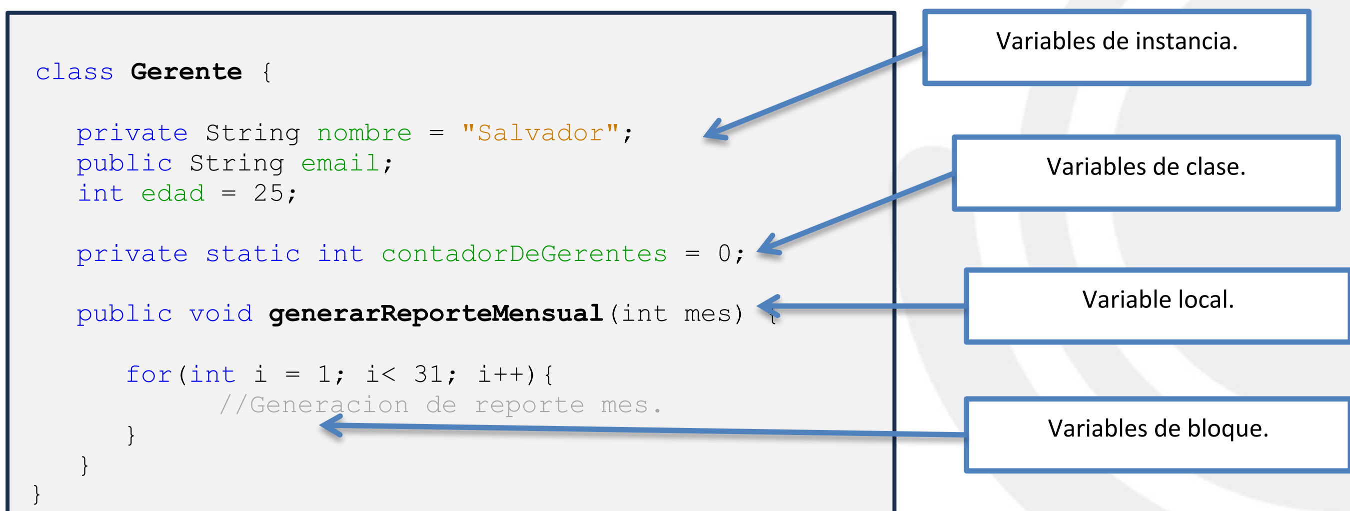
```
      float  int32 = 5.67f ;
      //Aplicando casting
      int   int32= (int)float32; // El valor que contiene int32 es  5.

      short  short16 = 128;
      byte   byte8 = (byte)short16;
      // el bloque byte solo soporta hasta 127 , por lo cual el valor se pasó en 1 
unidad y eso indica que ocupara un bloque más empezando por los números negativos 
contiendo el valor de -128.
```

2.5 Ámbito de las variables.

El ámbito de las variables no es más que otra cosa que el alcance de las variables (scope) y los diferentes tipos de variables que existen en java, indica por cuanto tiempo la variable va existir, existen cuatro tipos de ámbitos.

- **Variables de instancia:** Esta aparecen al crear una instancia de una clase y son inicializadas y existen mientras la instancia viva.
- **Variables locales**: Son aquellas declaradas dentro de un método incluyendo los parámetros que recibe de entrada, estas existen mientras el método este en ejecución, las características principales son que deben ser inicializadas de forma explícita antes de ser usadas y no pueden llevar modificador de acceso.
- **Variables de bloque:** son aquellas definidas por lagunas estructuras y solo viven mientras el bloque de código se esté ejecutando, un claro ejemplo seria las variables que se declaran dentro de un for, al igual que las locales no podrán usar un modificador deben ser inicializadas de forma explícita antes de utilizarlas.
- **Variables de clase**: también como variables estáticas y es creada cuando la clase es cargada, y solo existe mientras siga cargada la clase en la JVM, se debe tener cuidado ya que el hecho de esto es que solo existe una por clase y no de instancia, pero al igual que las de instancia son inicializadas de forma automática salvo que lleva el modificador de no acceso ***static***.

El siguiente código muestra el ámbito de las variables.



Al igual que los métodos y las clases, las variables en Java deben tener un nombre que sirve como identificador de las mismas y desde luego no se pueden definir dos o más variables con el mismo nombre.

Las convenciones para el nombrado de variables pueden comenzar con una letra, un guión bajo (_) o un signo de dólar ($) o de peso si se prefiere. Después del primer carácter sí es válido usar los símbolos de los 10 dígitos. Sin embargo, no se recomienda usar el guión bajo o el signo de dólar porque cuando el Java genera nombres, lo hace precisamente usando estos caracteres especiales como primer símbolo, de manera que si los desarrolladores no los usamos, no puede haber la posibilidad de duplicación. Se recomienda usar nombres descriptivos que sean fáciles de entender, por ejemplo, un nombre adecuado sería tasaDeInteres, mientras que TDI aunque válido no sería muy recomendable.

No se pueden usar otros caracteres especiales que no sean el guión bajo y el signo de dólar y éstos con las salvedades mencionadas.

Por ejemplo cantidadDe$ no es un identificador válido.

En la especificación del lenguaje Java no se restringe la longitud que puedan tener los nombres, pero desde luego no es recomendado usar nombres demasiado largos porque hacen el programa más grande y lento en su compilación. Tampoco se recomienda, por el otro lado, usar nombres muy pequeños por cuestiones de claridad.

Finalmente, los identificadores no pueden ser palabras reservadas. Java, como la mayoría de los lenguajes de programación, usa un juego de palabras que no permite que se usen como identificadores. En la siguiente lámina se muestran las palabras reservadas de Java.

develop
talent & technology

MÓDULO 03

Creación y uso de objetos

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

3.1 Variables de referencia a objetos

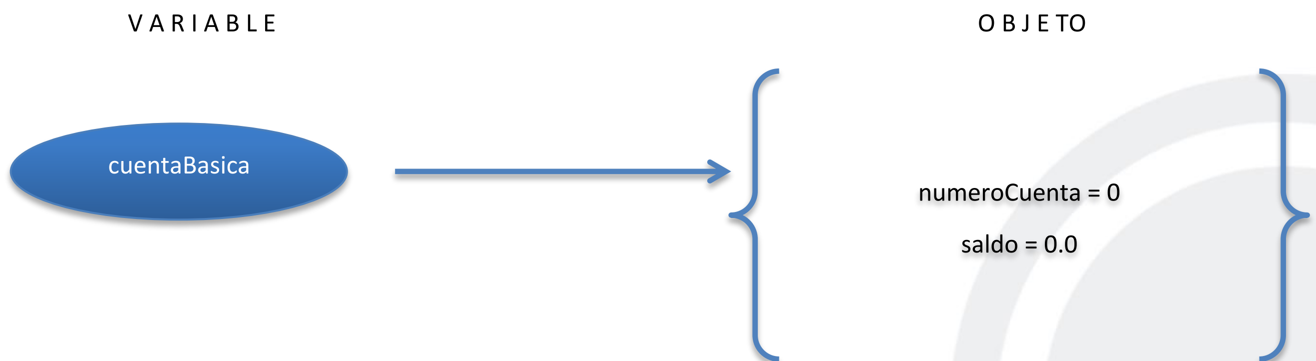
Antes de comenzar el tema de variables de referencia, es necesario no confundirlo con el ámbito de la variable ya que el ámbito solo indica alcance y en cualquier ámbito puede contener 2 tipos de variables tanto primitivos que existen 8 dadas por palabras reservadas que se cubrió en el módulo anterior, así como los tipos de referencia que se cubrirá en este módulo.

Un punto importante cuando se habla de ámbito, contexto o alcance específicamente las de instancia y estáticas o de clase son inicializadas por default y estas pueden llevar un nivel de acceso, mientras que las variables locales y de bloque deben ser inicializadas antes de su uso de lo contrario marca un error de compilación, tener cuidado con este punto ya que el hecho que se declare una variable local o de bloque no marca error si no al momento de ser utilizada en una operación y otra característica de estos dos últimos ámbitos es que no pueden llevar un nivel de acceso.

Para entender el concepto de variable de referencia, se examina el ejemplo del banco, específicamente sabemos que si tenemos una clase llamada CuentaBancaria y que tiene un conjunto de atributos y métodos y se crea una instancia es mucho fácil manipularla mediante una variable que hace referencia a sus atributos y métodos.

```
public class CuentaBancaria {
      public String numeroDeCuenta;
      public double saldo;
}
```

En donde el siguiente ejemplo muestra como una variable hace referencia a los atributos de un objeto, más adelante se explica por qué numeroCuenta y saldo tienen valores.



3.1.1 Declaración de variables de referencia

La sintaxis básica de la declaración de una variable de referencia se denota por contener un modificador de acceso, un tipo de dato, seguido de un identificador de variable valido como en el caso que se muestra cuando se tratan de variables de instancia.

En el siguiente código muestra ahora como la clase CuentaHabiente declara 3 tipos de variables de referencia en donde puede ser una creada por el desarrollador para el caso de CuentaBancaria o una ya existente en la API de java, en el siguiente modulo se explicara a detalle porque un CuentaHabiente contiene una CuentaBancaria o viceversa.

```
public class CuentaHabiente {
//Variables de instancia
      public String nombre;
      public System direccion;
      public CuentaBancaria cuenta;
}
```

Otra forma en que podemos declarar variables de referencia pero en el ámbito local es exactamente de la misma manera pero la diferencia es que no llevan modificador de acceso, de lo contrario marca error de compilación.

```
public class TestBanco {
      public static void main(String[] args) {
            //Variables locales: /*Sin modificador de acceso*/
            CuentaBancaria cuentaBasica;
            CuentaHabiente usuarioBasico;
      }
}
```

Un punto importante que debes tomar en cuenta es que el hecho de que se declaran 2 variables de referencia no significa que sean objetos, simplemente son variables que guardaran una referencia a un objeto que contiene los atributos y métodos para accederlos más tarde.

3.1.2 Instanciación de objetos

El siguiente paso es la instanciación que no es más que otra cosa que la creación del objeto, esta denotada por la parte después de la igualdad, cuya sintaxis está dada por la palabra reservada **new** seguido del tipo de dato, seguido de un par paréntesis que abren y cierran que denotan parámetros de entrada y finalizando con un punto y coma.

La sintaxis básica corresponde a la siguiente figura:

```
Sintaxis:
   Tipo nombreDeVariable = new Tipo([argumentos]);
```

Al llegar a este segmento de código se puede observar que ahora las variables cuentaBasica y usarioBasico ya se encuentran instanciadas correctamente.

```java
public class TestBanco {
      public static void main(String[] args) {
            /*Declaracion*/
            CuentaBancaria cuentaBasica;
            CuentaHabiente usuarioBasico;
            /*Instanciación*/
            cuentaBasica = new CuentaBancaria();
            usuarioBasico = new CuentaHabiente();
      }
}
```

3.1.3 Inicialización de variables de referencia

La inicialización no es más que otra cosa que la interpretación de la asignación, para ello la asignación recordar que se lee de derecha a izquierda, y también mencionar que el operador new, tiene dos principales tareas la primera es reservar espacio de memoria del tipo indicado y después de esa reservación devolver la referencia de donde vive el objeto para que posteriormente sea asignada a la variable. Como se muestra a continuación.

Sintaxis:

```java
   Tipo nombreDeVariable = new Tipo([argumentos]);
```

Es por esa razón que hasta este punto se llama variable de referencia ya que contiene la referencia de donde vive ese objeto y mediante esta referencia poder acceder a los atributos de ese objeto, se debe tomar en cuenta que un objeto es aquel que sus atributos ya tiene un estado es decir están inicializados o contienen un valor, por ejemplo en el siguiente código muestra como la cuentaBasica utiliza los atributos sin marcar error de compilación así como el usuario básico.

```java
public class TestBanco {
      public static void main(String[] args) {
            /*Declaracion*/
            CuentaBancaria cuentaBasica;
            CuentaHabiente usuarioBasico;
            /*Instanciación*/
            cuentaBasica = new CuentaBancaria();
            usuarioBasico = new CuentaHabiente();

            System.out.println("Nombre: " + usuarioBasico.nombre);
            System.out.println("Cuenta: " + usuarioBasico.cuenta);
            System.out.println("Saldo: " + cuentaBasica.saldo);

      }
}
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

Si se corriera la anterior aplicación la salida sería algo similar a esto:

```
Nombre: null
Cuenta: null
Saldo: 0.0
```

La razón por las que se tiene esa salida es debido a que la variable de referencia usuarioBasico hace referencia al objeto de tipo CuentaHabiente y como es un objeto sus atributos o variables de instancia se encuentra inicializadas de forma automática debido a que ya tienen un estado y al detectar que el nombre de tipo String y cuenta de tipo CuentaBancaria son referencias y estas serán inicializadas por default en null, mientras que la variable cuentaBasica que hace referencia a un objeto de tipo CuentaBancaria y tiene un atributo llamado saldo este es inicializado por default en 0.0.

La siguiente tabla detalla la inicialización por default de acuerdo al tipo de dato:

Tipo de dato	Valor
byte	0
short	0
int	0
long	0
float	0.0f
double	0.0
char	'\u0000'
boolean	false
***Referencia(Clases)**	null

3.2 Cómo se almacenan en memoria las variables de referencia

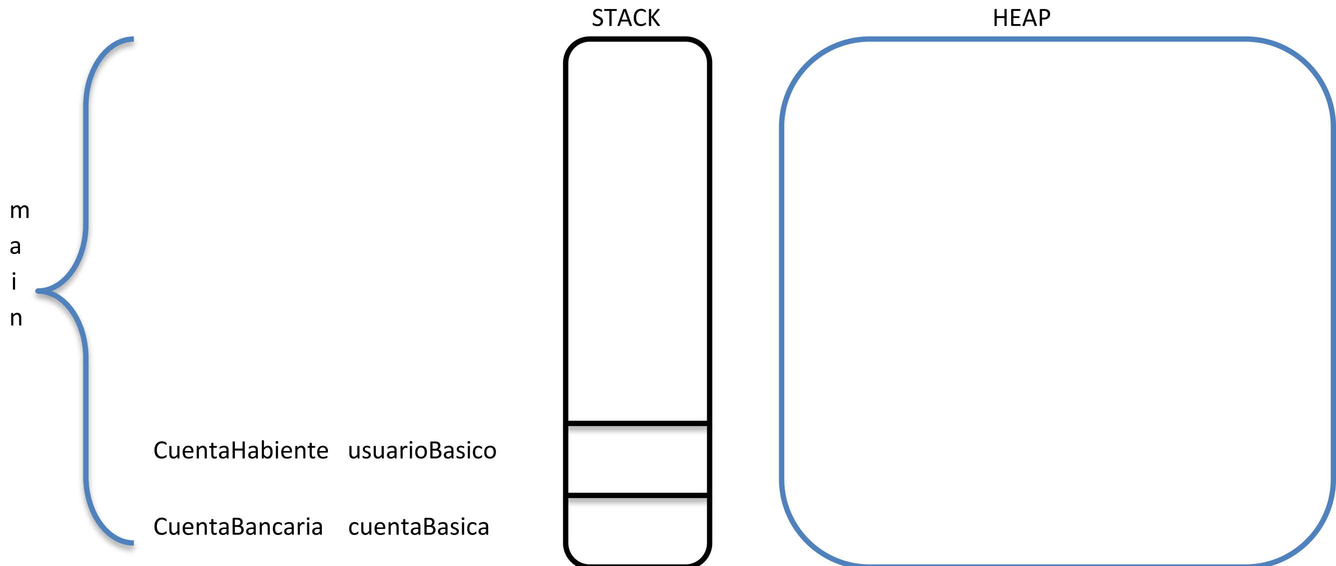
La JVM para la correcta funcionalidad de una aplicación contiene una área de memoria que está dividida en dos una llamada Stack el cual no es más que una pila con estructura FIFO (First In First Out) que el principal objetivo es indicar las tareas de la aplicación que se está llevando a cabo en orden conforme se manden a llamar, no olvidar que el comienzo de una aplicación es por el método principal, bien esta pila es la encargada de almacenar las variables de esos método (variables locales y variables de bloque); mientras que la otra parte es llamada Heap el cual es el encargado de almacenar los objetos con sus respectivas variables de instancia.

El siguiente código mostrara el diagrama en Stack-Heap.

```
public class TestBanco {
      public static void main(String[] args) {
            //Variables locales: /*Sin modificador de acceso*/
            CuentaBancaria cuentaBasica;
            CuentaHabiente usuarioBasico;
      }
}
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

Una aplicación en java comienza con la carga del método principal y sus respectivas variables locales en stack, las cuales son args, cuentaBasica y usuarioBasico, pero al no ver el tema de arreglos a fondo que se cubre en el módulo 8 se omitirá la variable args, a continuación el diagrama.



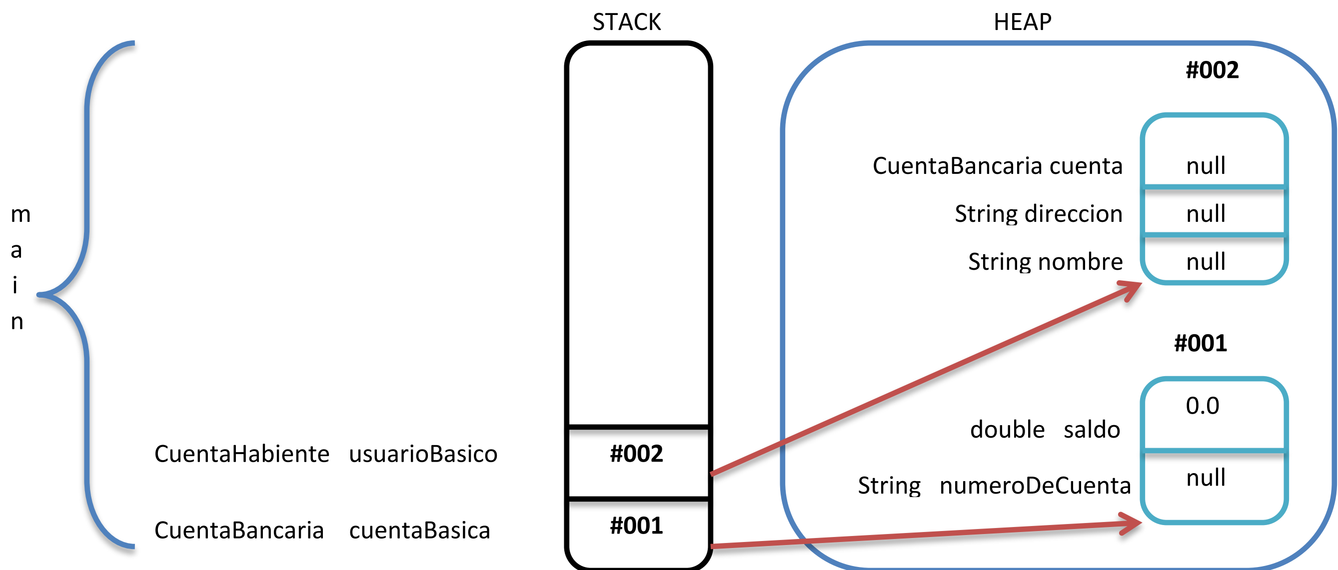
Como se puede observar no se ha construido ningún objeto, simplemente las variables locales se encuentran sin inicializar. El siguiente código muestra el momento en el cual se realiza la instancia y además se inicializan.

```
public class TestBanco {
      public static void main(String[] args) {
            /*Declaracion*/
            CuentaBancaria cuentaBasica;
            CuentaHabiente usuarioBasico;
            /*Instanciación*/
            cuentaBasica = new CuentaBancaria();
            usuarioBasico = new CuentaHabiente();
      }
}
```

Una vez declaradas las variables, el siguiente paso es la instanciación, para ello se resume a los siguientes pasos:

- Reservar espacio de memoria del tipo de dato indicado en el área de Heap.
- Colocar las variables de instancia en el espacio reservado del tipo de dato reservado
- Inicializar las variables de instancia por default.(Véase tabla de inicialización por default)
- Una vez que la inicialización fue realizada correctamente, al espacio reservado se le da una dirección de memoria.

- Se interpreta la asignación, la dirección de memoria generada por el operador new es asignada al variable de referencia que se encuentra en pila.
- Finalmente la variable en pila deberá hacer referencia al objeto en heap (apuntar).



Es por esa razón que las variables usuarioBasico y cuentaBasica de llaman de referencia ya que hacen referencia al objeto que se encuentra en heap por el cual se puede acceder mediante la notación "." Siempre y cuando así lo permita el modificador de visibilidad.

```
public class TestBanco {
      public static void main(String[] args) {
            /*Declaracion*/
            CuentaBancaria cuentaBasica;
            CuentaHabiente usuarioBasico;
            /*Instanciación*/
            cuentaBasica = new CuentaBancaria();
            usuarioBasico = new CuentaHabiente();

            System.out.println("Nombre: " + usuarioBasico.nombre);
            System.out.println("Cuenta: " + usuarioBasico.cuenta);
            System.out.println("Saldo: " + cuentaBasica.saldo);

      }
}
```

usuarioBasico mediante la referencia #002 se accede al nombre y se observa que es null

cuentaBasica mediante la referencia #001 accede al saldo y se observa que equivale a 0.0

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

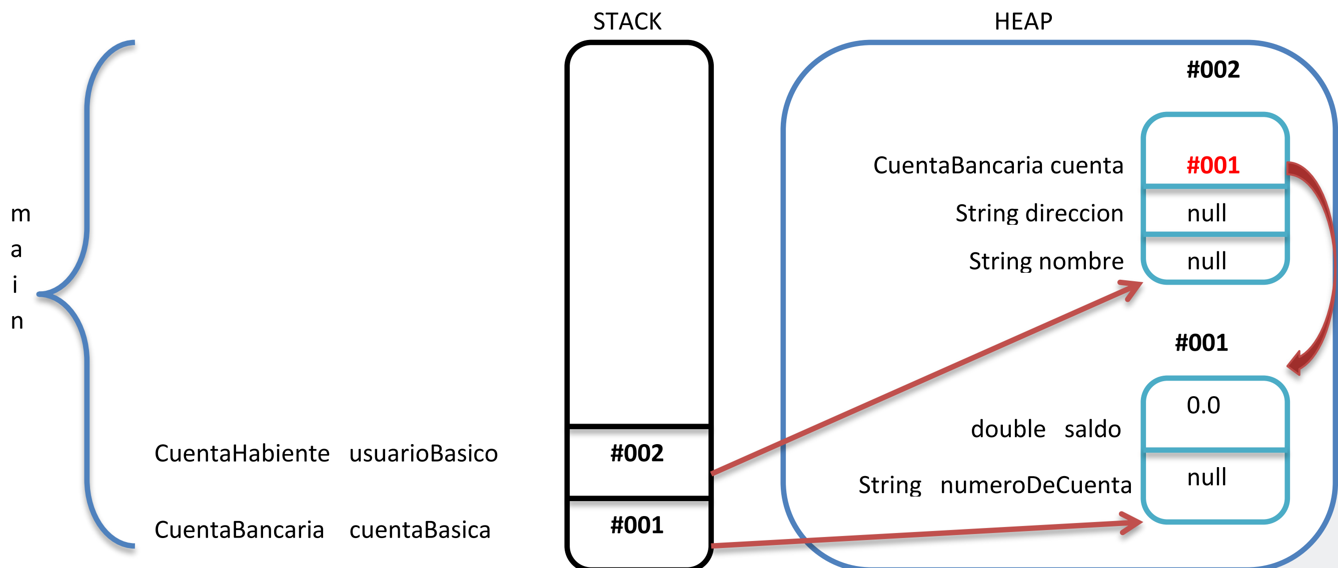
3.3 Manejo de objetos y variables de referencia

Come se menciona en el punto anterior se ha manipulado los objetos por separado cada variable hace referencia al objeto, pero también se pueden asignar otras referencias al objeto siempre y cuando sean compatibles al igual que los valores primitivos.

En la figura siguiente se muestra como mediante la referencia de usuarioBasico accede al atributo cuenta para asignarle la referencia de cuentaBasica, esto se permite ya que son del mismo tipo de dato, en forma de asignación se leería que la referencia de cuentaBasica es asignada a la variable cuenta de tipo CuentaBancaria referenciado por la variable usarioBasico.

```
public class TestBanco {
      public static void main(String[] args) {
            CuentaBancaria cuentaBasica = new CuentaBancaria();
            CuentaHabiente usuarioBasico = new CuentaHabiente();
            usuarioBasico.cuenta = cuentaBasica;

      }
}
```

Representado en diagrama se visualiza de la siguiente manera donde el atributo cuenta de usuarioBasico hace referencia a donde vive el objeto referenciado por la variable cuentaBasica:

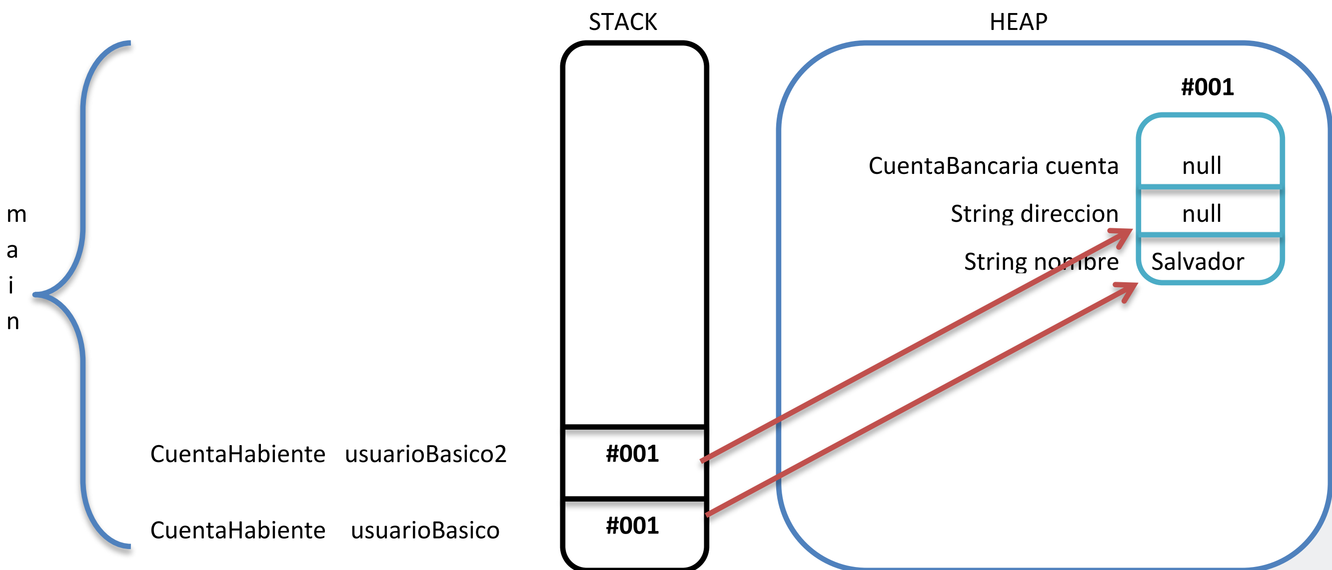


Otra manera de representar el manejo de objetos es mediante otra variable de referencia del mismo tipo, esto es muy usado cuando se pasa la referencia como parámetro de un método, verdaderamente pasas la referencia del objeto, en el siguiente código se muestra como 2 variables pueden contener o apuntar al mismo objeto.

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

```java
public class TestBanco {
	public static void main(String[] args) {
		CuentaHabiente usuarioBasico = new CuentaHabiente();
		CuentaHabiente usuarioBasico2 = usuarioBasico;

		usuarioBasico.nombre = "Salvador"
		System.out.println("Nombre: " + usuarioBasico.nombre);
		System.out.println("Nombre: " + usuarioBasico2.nombre);
		/*Ambas variables contienen el mismo nombre de Salvador*/
	}
}
```

En esta circunstancia observa que ahora 2 variables hacen referencia al único objeto, con cualquiera de la variables puede cambiar el estado los atributos, esto quiere decir que los datos pueden ser modificados utilizando usuarioBasico o usarioBasico2

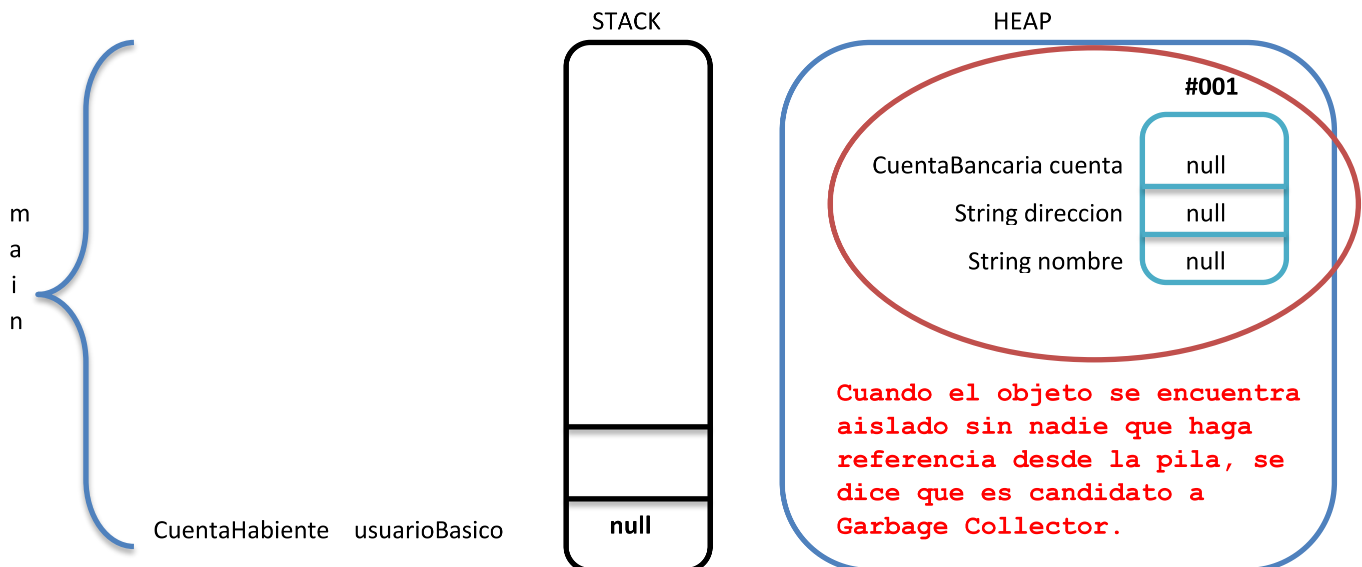


3.3.1 Garbage Collector

En el siguiente ejemplo para finalizar este modulo se hablara de lo que es el recolector de basura (Garbage collector) el cual es un mecanismo que permite la liberacion de la memoria de los objetos que ya no se necesiten en Heap para la creacion de nuevos objetos porteiores, este mecanismo permite pasar una vez limpiando la memoria de forma automatica teniendo como desventaja pasar la memoria cuando ejecutarse y cuando no, podemos sugerir que tambien de forma personalizada pase el recolector de basura pero no garantiza que en verdad se solcite una recoleccion de memoria.

```java
public class TestBanco {
	public static void main(String[] args) {
		CuentaHabiente usuarioBasico = new CuentaHabiente();
		usuarioBasico = null;
	}
}
```

Por ejemplo en el esquema se muestra como el objeto ya no es referenciado por la variable si no que ahora contiene un valor de nulo y ese objeto se dice que es candidato al Garbage Collector como se muestra en el siguiente ejemplo.

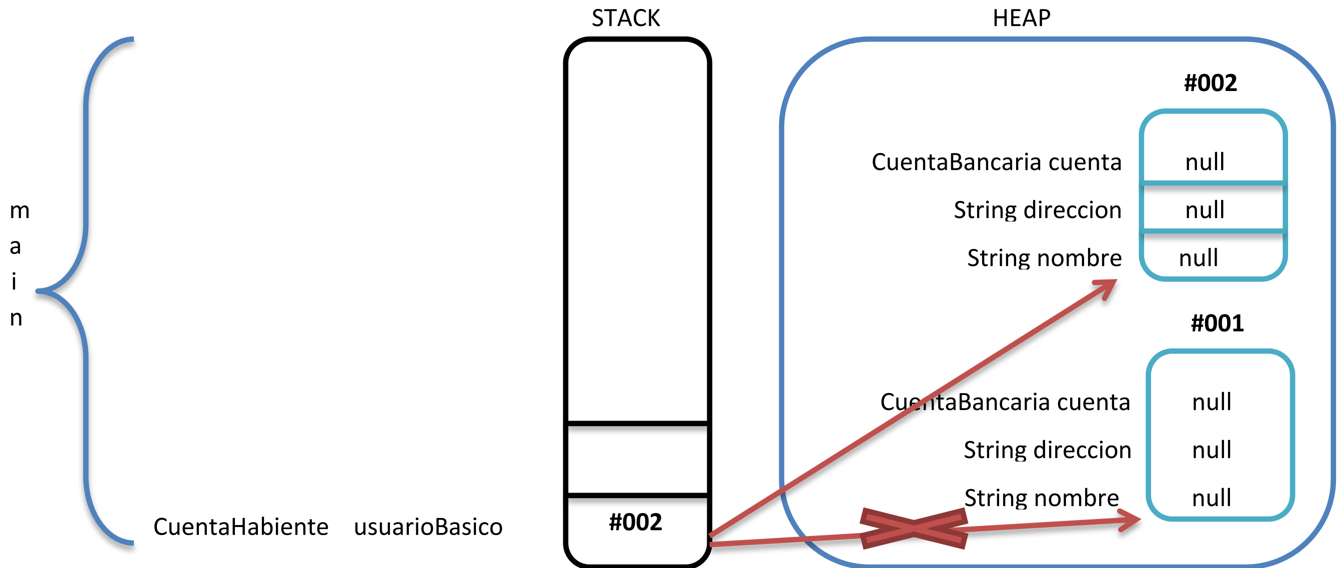


Aunque también es posible perder la referencia creando un nuevo objeto, ya que solo la variable puede contener una referencia a la vez.

El siguiente diagrama puede demostrar una segunda forma de perder la referencia al objeto y apuntar un nuevo objeto.

```java
public class TestBanco {
	public static void main(String[] args) {
		CuentaHabiente usuarioBasico = new CuentaHabiente();
		usuarioBasico = new CuentaHabiente();
	}
}
```

Ahora el objeto referenciado por #001 es candidato a Garbage Collector, mientras que usuarioBasico contiene una referencia #002 al nuevo objeto.



develop
talent & technology

MÓDULO 04

Operadores y construcciones de toma de decisión

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

4.1 Uso de operadores relacionales y condicionales

4.1.1 Operadores relacionales

Los operadores relacionales son aquellos que comparan valores para ver igualdad o desigualdad y su principal característica es que siempre producirán un resultado de tipo bolean. A continuación, se enlistan la tabla de operadores que comparan relaciones y que existen en java:

Operador	Significado
<	Menor que
<=	Menor o igual que
>	Mayor que
>=	Mayor que
==	Igual que
¡=	Diferente de

El siguiente ejemplo describe una manera de utilizar los operadores y utilizando orden de precedencia.

```
public static void main(String[] args) {
      int valor = 2;
      System.out.println("Resultado: " + ( 6 == (valor = ++valor * 4) ));
}
```

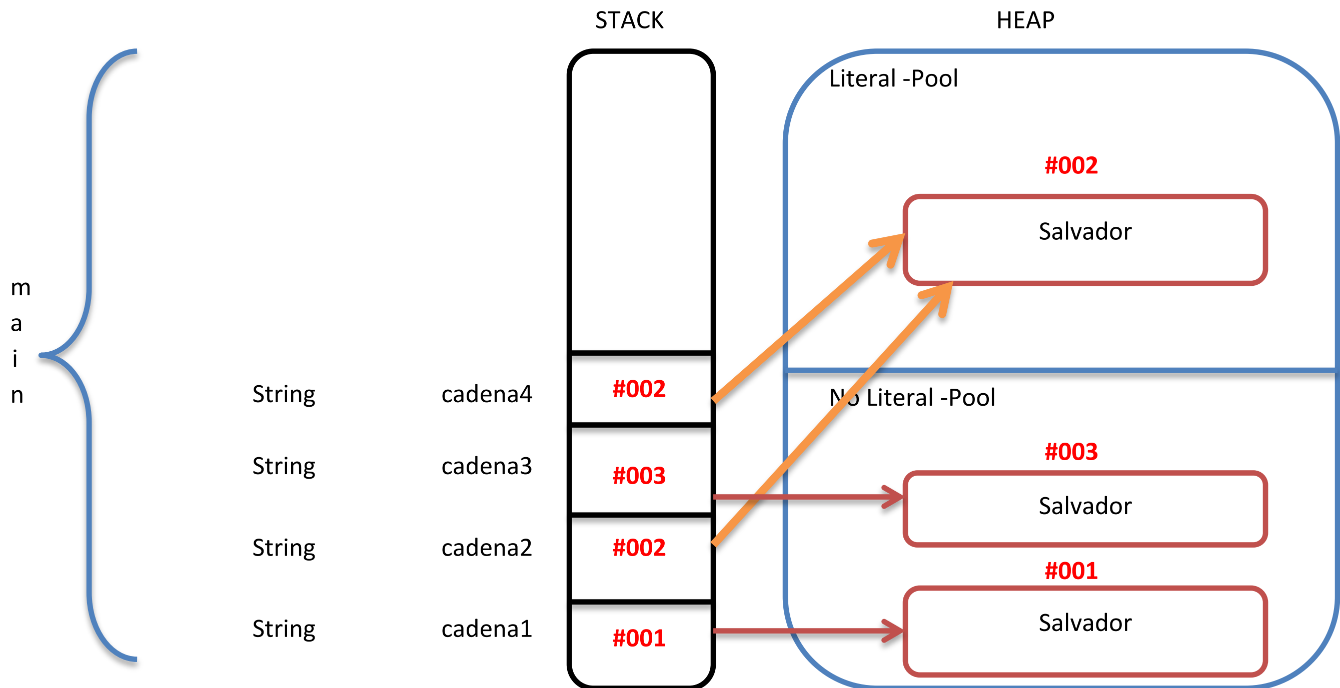
Como primer paso se debe realizar todo lo que se encuentra entre paréntesis:

1. Evalúa la expresión (`valor = ++valor * 3)` en esta expresión se tiene que tener cuidado ya que existe una asignación que es evaluada hasta el final y este no compara ningún valor. No confundir = `e ==`.
2. Una vez evaluada el resultado, en donde primero incremento una unidad valiendo 3 y utilizada por lo operación como 3 *4 y ese valor es asignado a valor contiendo un 12.
3. Finalmente se interpreta la comparación en donde pregunta si 6 ex exactamente a 12, cuyo resultado es Resultado: false.

Cuando se están comparando variables de referencia con el operador == o ¡= son las únicas formas validas ya que no se puede ver la relación de >, >=, <, <= y estas marcan error de compilación. Las clases String que son tipos de referencia se observa que solo se compara lo que se encuentra almacenado es Stack como se muestra a continuación:

Ejemplo:
```
      String cadena1 = new  String("Salvador");
      String cadena2 = "Salvador";
      String cadena3 = new  String("Salvador");
      String cadena4 = "Salvador";

      System.out.println(cadena1 == cadena2); // false
      System.out.println(cadena2 == cadena3); // false
      System.out.println(cadena4 == cadena2); // true
```

Visto desde el diagrama Heap-Stack:



4.1.2 Operadores condicionales

Los operadores condicionales son aquellos que permiten realizar una relación más compleja apoyándose en compuertas lógicas, la siguiente tabla muestra los operadores condicionales que existen en java.

Operador	Ejemplo	Significado
&&	Condicion1 && condicion2	Operador de corto circuito. Si la primera condición es un false ya no se evaluará la segunda condición. todas las condiciones deben ser verdaderas para que el resultado sea verdadero. (AND)
||	Condicion1 || condicion2	Operador de corto circuito. Si la primera condición es un true ya no se evaluará el resto de las condiciones, resultado true. Por lo menos una condición debe ser verdadera para tener un true. (OR)
&	Condicion1 & condicion2	Operador de circuito completo. Evalúan todas las condiciones. (AND)
|	Condicion1 | condicion2	Operador de circuito completo. Evalúan todas las condiciones. (OR)
^	Condicion1 ^ condicion2	Evalúa dos términos, si son diferentes el resultado es false, si son iguales es true.
¡	¡Condicion1	Evalúa solo un término y el resultado es el valor de entrada invertido.

La siguiente tabla describe las tablas de verdad de las compuertas lógicas

Ent1	Ent2	Salida
0	0	0
0	1	0
1	0	0
1	1	1

Ent1	Ent2	Salida
0	0	0
0	1	1
1	0	1
1	1	1

Ent1	Ent2	Salida
0	0	0
0	1	1
1	0	1
1	1	0

Ent1	Salida
0	1
1	0

El siguiente ejemplo muestra el uso del operador de corto circuito utilizando la compuerta && (AND).

```
public class TestCortoCircuito {
      public static void main(String[] args) {
            int v1=10;
            int v2=2;
            System.out.println((v1++ == ++v2) && (++v1 == v2++));// false
            System.out.println("v1 = " + v1); // v1 = 11
            System.out.println("v2 = " + v2); // v2 = 3
      }
}
```

Ent1	Ent2	Salida
0	0	0
0	1	0
1	0	0
1	1	1

En el ejemplo anterior la primer expresión a evaluar es la parte izquierda donde a v1 es usada en la comparación como 10 pero pasando a la siguiente expresión ya vale 11, y en cuanto el valor de v2 en la primer condición indica que primero se incremente a 3 y es usado en la comparación como 3, al final quedando la comparación como 11 == 3, conociendo su resultado false, al observar la tabla de verdad que en todos los casos cuando el primer término es un false sin importar evaluar la segunda expresión el resultado es false, por lo tanto corta el circuito y no evaluara el segundo término después de la compuerta se conoce el resultado a un false, al final los valores para v1 es 11 y v2 es 3.

El siguiente ejemplo muestra el uso del operador de circuito completo utilizando la compuerta & (AND).

```
public class TestCircuitoAbierto {
      public static void main(String[] args) {
            int v1=10;
            int v2=2;
            System.out.println((v1++ == ++v2) & (++v1 == v2++)); // false
            System.out.println("v1 = " + v1); // v1 = 12
            System.out.println("v2 = " + v2); // v2 = 4
      }
}
```

En este ejemplo si está obligado a evaluar ambos términos quedando como resultado para la variable v1 el valor de 12 y de v2 el valor 4.

Fundamentos de Programación en Java | Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

4.2 Sentencia *de toma de decisión*

4.2.1 Sentencia *if*

La sentencia está dado por la palabra reservada if, seguido de un par de paréntesis que abren y cierran y dentro de ella una expresión booleana que puede ser generado ya sea de una operación condicional o relacional, pero que al final contenga una expresión booleana.

Sintaxis:
```
if (expresion_booleane){
      /*alcance del bloque if*/
};
```

A partir de esta sentencia el alcance de las actividades está dada por un bloque de código interpretado por un par de llaves, pero si la sentencia no contiene un par de llaves el alcance de esta estructura será la primera línea de código o sentencia y se debe tener cuidado como es utilizada.

Los siguientes ejemplos muestra algunas formas de ver la sentencia if:

Ejemplo:
```
public class TestIf {
      public static void main(String[] args) {
            int v1 = 10;
            boolean v2 = true;

            if ( v1 == 10){}
            if ( v2 == true){}
            if ( v2 = true){}
            if ( v1 = 10){}
      }
}
```

El caso anterior todos compilan excepto la sentencia if **(v1 = 10){}** marca un error de compilación debido a que al asignar el 10 a v1, el **if** no puede evaluar la expresión booleano, es como tener **if (10){}** el cual no es un valor booleano.

Ahora cabe mencionar que un if puede contener dentro de su bloque otro if, a esto se le conoce como if anidado.

4.2.2 Sentencia *if-else*

La sentencia if-else, es utilizada cuando en el flujo se tienen 2 posibles caminos y en cada caso se realizan actividades diferentes, la estructura es que debe ir en seguida de la sentencia if de lo contrario será un error de compilación y además esta no debe llevar condición, utiliza la misma dinámica que un if, de no llevar bloque de código el alcance de la estructura es la primera sentencia que se encuentra enseguida, a continuación, el ejemplo:

Ejemplo:
```
 public class TestIfElse {
      public static void main(String[] args) {
            boolean v2 = true;

            if ( v2 == true){
                  /*procesamiento en caso verdadero*/
            }
            else {
                  /*procesamiento en caso falso*/
            }
      }
}
```

4.2.3 Sentencia *if-else-if*

Y finalmente la sentencia if-else-if, esta estructura a diferencia de la anterior es que lleva condición y esta denotada por la composición de 2 palabras reservadas **else if** y debe en seguida de la declaración de la sentencia **if** pero antes de la sentencia **else**, además de que puede contener N declaraciones de esta estructura siempre y cuando sea después del if y antes del else.

Ejemplo:
```
 public class TestIfElseIf {
      public static void main(String[] args) {
            int x1 = 4;

            if ( x1 == 2){ /*procesamiento en caso de ser 2*/}
            else if ( x1 == 3){/*procesamiento en caso de ser 3*/}
            else if ( x1 == 4){ /*procesamiento en caso de ser 4*/}
            else { /*procesamiento en caso distinto*/}
      }
}
```

4.3 Sentencia *switch*

La estructura switch, fue creada para tener relación de varios casos que tienen el procesamiento en común, una versión alternativa del if-else-if. La sintaxis básica de un switch está dada por:

Sintaxis:
```
 switch (opcion){
      case constante:
            /*actividades según el caso ;*/
 }
```

Entre los tipos de datos que son soportados en switch son: todos los tipos de datos por default de un int (byte, short e int), incluyendo el char, tipos de datos enumerados y a partir de la versión 7 de java es permitido la clase String pero no otros tipos de datos de referencia, ni tampoco boolean, ni tipos de datos que manejen punto flotante.En cuantos a los valores de los casos no se pueden colocar variables, si no constantes debido a que son casos a ser evaluados en comparación de forma estática más no de forma dinámica.

Ejemplo:

```
 public class TestSwitch {
      public static void main(String[] args) {
            int opcion = 3;
            final int MY_CASE = 4;
            switch ( opcion){
                  case 2:/*procesamiento en caso de ser 2*/
                  case 3:/*procesamiento en caso de ser 3*/
                  case MY_CASE:/*procesamiento en caso de ser 3*/
            }
}
```

Si se interpretara el código anterior, MY_CASE es una constante indicado por el modificador final lo cual es permitido ser utilizado en un case de un switch; el resultado sería el procesamiento tanto de case 3 y case 4, debido a que diferencia del if estas continuaran con los siguientes casos, es por ello que se al final del caso opcionalmente puede colocarse la sentencia break para forzar a terminar toda la estructura switch.

4.3.1 Cláusula *break.*

La cláusula break como se menciona anteriormente es utilizada para finalizar la estructura switch, aunque también puede servir para terminar una estructura de ciclo, que se verá en el siguiente modulo.

```
public class TestSwitch {
      public static void main(String[] args) {
            int opcion = 3;
            final int MY_CASE = 4;
            switch ( opcion){
                  case 2:
                  /*procesamiento en caso de ser 2*/
                  break;
                  case 3:
                  /*procesamiento en caso de ser 3*/
                  break;
                  case MY_CASE:
                  /*procesamiento en caso de ser 3*/
                  break;
            }
}
```

4.3.2 Cláusula *default.*

En dado caso que el valor o la opción no coincida en alguno de los casos, la sentencia switch no se ejecuta, es para ello que se crea un case especial dado por la palabra reservada **default** que evaluara cualquier case no especificado es por eso que este no lleva una constante solamente es la palabra default seguido de dos puntos y en seguida el procesamiento, opcionalmente puede llevar el break, pero no es necesario si es último caso de un switch.

Ejemplo:

```
 public class TestSwitch {
      public static void main(String[] args) {
            int opcion = 8;
            final int MY_CASE = 4;
            switch ( opcion){
                  case 2:
                  /*procesamiento en caso de ser 2*/
                  break;
                  case 3:
                  /*procesamiento en caso de ser 3*/
                  break;
                  case MY_CASE:
                  /*procesamiento en caso de ser 3*/
                  break;
                  default:
                  /*procesamiento en cualquier otro caso distinto a los
                  declarados*/
            }
}
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

MÓDULO 05

Manejo de ciclos

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

5.1 Creación de ciclos *while*

La sentencia while es ideal cuando necesitamos que se ejecute un boque de código tantas veces sea necesario pero que en algún momento termine.

Las características del while se describen a continuación:

- Siempre que la condición sea verdadera se ejecuta el bloque de código.
- Si no contiene llaves el alcance de la estructura while será la primer sentencia o línea de código subsecuente.
- Si la expresión es inicialmente falsa no se ejecutará nunca el bloque.

Sintaxis:
```
 while (/*expresion_boolean*/){
             /*actividades */
 }
```

5.2 Creación de ciclos *for*

La estructura for es un poco más compleja, ya que en sus estructura lleva tres secciones separadas por punto y coma, incluyendo la de la condición como se muestra en la sintaxis.

Sintaxis:
```
for (/*inicializacion*/ ; /*expresion_boolean*/ ; /*varianza*/){
             /*actividades */
}
```

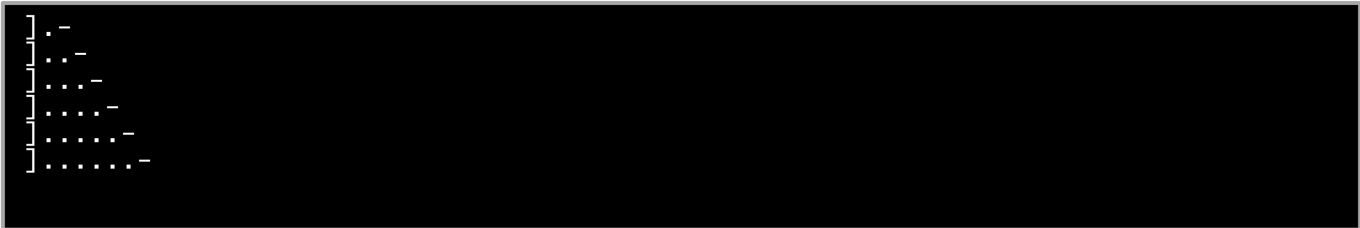
- Si no se especifica nada en cualquiera de los 3 apartado del for, la condición siempre será verdadero y nunca termina (bucle infinito).
- En el apartado de la inicialización solo se ejecuta una vez en todo el ciclo de vida del for y solo se permite inicializar un solo tipo de dato.
- En el apartado de inicialización y de varianza se pueden colocar más instrucciones separadas por comas.
- En el apartado de inicialización y de varianza no solo sirve para tener variables, sino que también sirven para declarar otro tipo de sentencias como por ejemplo una instrucción de impresión.
- Si no contiene llaves el alcance de la estructura for será la primer sentencia o línea de código subsecuente.
- Si se declaran variables en el apartado de inicialización solo puede ser utilizado en el bloque del for y solo es permitido declarar varias variables del mismo tipo de datos.
- Pero si es posible inicializar otros variables de diferente tipo

A continuación, un ejemplo de cómo definir la estructura de un for clásico, describiendo lo antes mencionado.

Fundamentos de Programación en Java | Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

Ejemplo:
```
int alfa =  5;
String x;

for(alfa=0,x="]";alfa<64; alfa=alfa*2,System.out.println("-")){
      alfa++;
      System.out.print(x=x+".");
}
```

Al final de la ejecución se tiene la siguiente salida:


```
].-
]..-
]...-
]....-
].....-
]......-
```

Existe una versión sobrecargado de la estructura for, conocida como foreach a partir de la versión 6 de java, esta estructura permite la iteración de toda la colección o un arreglo, a diferencia del for clásico está conformado por dos secciones separadas por el signo de dos puntos, en la parte derecha indica el arreglo o la colección, mientras que en la parte izquierda indica la variable que sostiene el elemento en cada iteración, debe ser declarado del tipo el cual es la colección o el arreglo, la siguiente figura muestra la sintaxis del for each.

Sintaxis:
```
for (/*Type aux*/ : /*Array/Collection */){
            /*actividades */
}
```

5.3 Creación de ciclos *do-while*

La estructura do-while es muy parecida a la estructura de while, la diferencia es que se ejecuta primero el código y después se evalúa la condición, caso contrario al while, otra gran diferencia es que el while debe terminar en punto y coma.

Sintaxis:
```
do{
    /*actividades */
}while(/*expresion_boolean*/);
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

Normalmente el do while es utilizando en situaciones donde se debe pedir un dato valido y dejar de pedirlo hasta que sea correcto, un claro ejemplo la generación de menús.

5.4 Comparación de los ciclos

Bucle	Uso Ideal
while	Cuando no se conoce la cantidad de elementos a iterar.
for clásico	Ideal cuando se conoce la cantidad a iterar, así como de un rango inicial y final.
foreach	Ideal cuando se requiere la iteración de todo un arreglo o colección.
do while	Ideal cuando se requiere la ejecución del bloque de código por lo menos una vez.

5.5 Ciclos anidados

La principal ventaja de usar bucles es la iteración de conjuntos de datos, pero muchas veces no es suficiente hacer la iteración correspondiente mediante un bucle, es aquí la necesidad de hacer ciclos anidados, el cual puede consistir en uno contener a otro.

Un ejemplo de donde puede verse un anidamiento de esos bucles es el siguiente caso, se sabe que un banco tiene un conjunto de cuentahabientes y un cuentahabiente tiene un conjunto de cuentas bancarias, ahora bien, si se tiene que generar una vista de impresión y solo se tiene permitido pasar como parámetro una instancia de un banco es ideal hacer un ciclo anidado, en donde el primer ciclo itere los cuentahabientes, y el segundo ciclo itera las cuentas de cada cuentahabiente.

La estructura a utilizar dependerá de las circunstancias, a continuación, un ejemplo de donde puede ser ocupado un bucle anidado.

Ejemplo:

```
public void generarReporte(Banco banco) {
	for(CuentaHabiente[] clientes : banco.getCuentaHabientes() ){
		for(CuentaBancaria cuenta : clientes.getCuentaBancarias() ){
			/*procesar el reporte de la cuentaBancaria por cliente*/
		}
	}
}
```

La razón por la que una estructura de control no lleva llaves es debido a que al notar en el caso anterior el for externo que itera los cuentahabientes del banco enseguida la instrucción es otro for, por lo cual no tendría sentido llevar llaves ya que es solo una instrucción a ejecutarse, aunque por buena práctica, aunque contenga solo una instrucción se debe colocar llaves que delimiten el alcance para una mayor legibilidad y rápido mantenimiento.

Ejemplo del for anterior delimitado sin llaves.

Ejemplo:
```
public void generarReporte(Banco banco) {
      for(CuentaHabiente[] clientes : banco.getCuentaHabientes() )
            for(CuentaBancaria cuenta : clientes.getCuentaBancarias() )
                  /*Si solo es una sentencia terminada en ;*/
}
```

Ahora bien, dentro del mismo tema de anidamiento en especial se tiene que tener cuidado cuando se manipulan do-while y while ya que pongamos un ejemplo en donde no existe error de compilación, pero es poco legible, es por ello el uso de la indexación de espacios para delimitar el alcance de cada bucle, debido a que solo se tiene una sentencia delimitada por punto y coma.

Ejemplo:
```
int x = 6;
int y = 7;
int z = 9;
do
      while(¡( z%33 == 0))
            z=x++*y++;
while(x++ != --y);
System.out.println("x = " + x); // x = 13
System.out.println("y = " + y); // y = 12
System.out.println("z = " + z); // z = 132
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

develop
talent & technology

MÓDULO 06

Arreglos

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

6.1 Arreglos unidimensionales

Es importante diferenciar los arreglos en Java con otros lenguajes, debido en el lenguaje no se maneja tanto el concepto de fila y columna, si no que se maneja el concepto de variables de referencia, es decir se tratan como objetos a diferencia de otros lenguajes, las características de un arreglo son tratados como forma homogénea es decir que solo se permitirá un solo tipo de dato y de forma estática, es decir que están limitada a una cantidad de elementos al momento de instanciarla, por lo cual esta permiten tener la asociación y multiplicidad de una a muchos.

Algo que cabe recordar es que si se trata de un arreglo como variable de instancia se inicializa por default en null, si se trata de un arreglo como variable local recordar que se debe inicializar de forma explícita antes de usarse de lo contrario marca un error de compilación.

Los arreglos normalmente son usados para referirse mediante a una variable a un grupo de elementos, que, a diferencia de otras circunstancias, es óptimo tener una variable que hace referencia a un conjunto de variables que tener muchas variables declaradas.

La sintaxis general de poder declarar e instanciar un arreglo se resume a la siguiente sintaxis.

Sintaxis:
```
type [] x = new type[CANTIDAD];
type x []= new type[CANTIDAD];
```

6.1.1 Arreglo de primitivos.

Los arreglos primitivos soportan solo valores del tipo declarado o en su defecto el valor previo, por ejemplo, si se declara un arreglo de enteros, podrá contener valores que fueron declarados como byte, short y char, para entender mejor este concepto es recomendable revisar la tabla del módulo 2 del tema promoción y casting.

En Java existen 2 formas de declaración:

Sintaxis:
```
int [] x;
int y[];
```

La principal diferencia en que existen estas dos formas de declaración es cuando se requiere declarar más variables en la misma línea, en el primer caso sirve para indicar que todas las variables serán de tipo arreglo, mientras que para la segunda forma indica que sola la primera declaración será un arreglo y las subsecuentes serán de tipo int.

Sintaxis:
```
int [] x,x1,x2; //las 3 variables son declaradas como arreglos.
int y[],y1; //solo y es declarado como arreglo, y1 es de tipo entero.
```

Al tratarse de un objeto en java cuando se refiere a los arreglos, es que se deben instanciar, pero a su vez las posiciones del arreglo son inicializadas de acuerdo a la tabla de inicialización, por ejemplo, si se realiza un arreglo de puntos de flotantes con 5 elementos esas posiciones serán inicializadas de forma automática en 0.0f, Existe 3 formas de inicializar e instanciar un arreglo.

1. **Operador new:** En esta forma se reserva espacio de memoria con la cantidad de elementos establecida que se indica entre corchetes, esta forma puede hacerse en dos sentencias, el punto importante que forzosamente debe indicarse entre los corchetes la cantidad de elementos, de lo contrario no compila.

```
Ejemplo 1:
short array[] = new short[5];

Ejemplo 2:
short array2[];
array2 = new short [5];
```

2. **Utilizando notación corchetes:** Esta forma es utilizada cuando se conoce la cantidad elementos y su respectivo valor y es por ello que al conocerse estos datos solo es permitido declarar e instanciar en la misma línea de sentencia de lo contrario será error de compilación.

```
Ejemplo 1:
boolean  array3[] = { false, true, false };
```

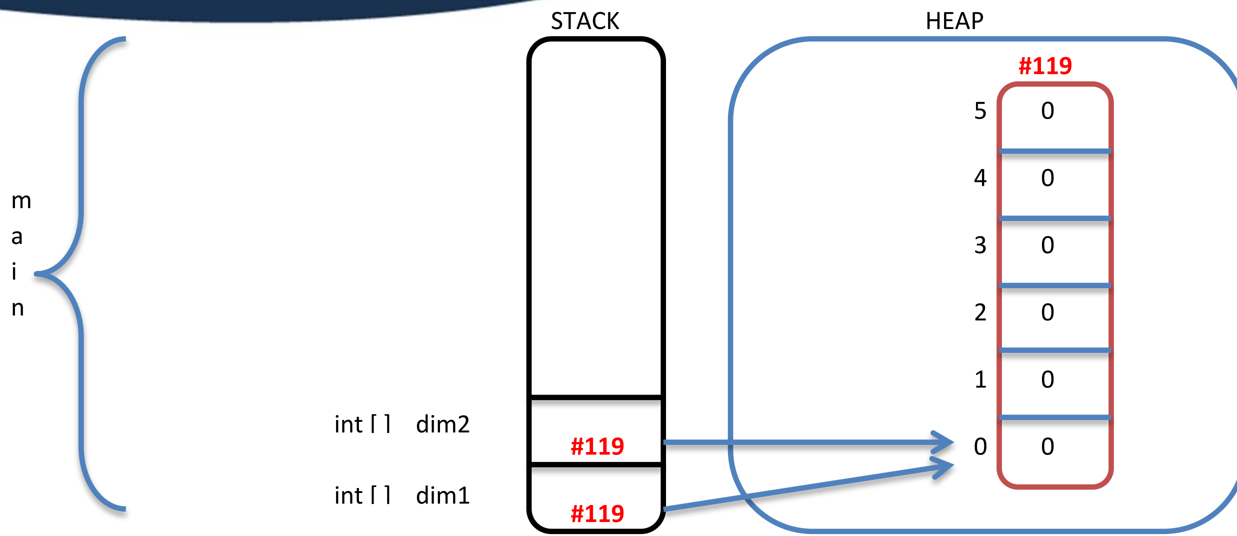
3. **Instanciando anónimamente:** Esta declaración es muy poco común, se diferencia porque es una combinación de las previas salvo por que en los corchetes no lleva cantidad de elementos si no que son inicializadas entre llaves, es muy útil cuando se genere al momento y enviarse como parámetro.

```
Ejemplo 1:
int  array4[] = { 5, 6, 7, 8,9, 10 };
```

Cualesquiera que sean las formas de instanciarse se representa de igual manera en memoria como se muestra a continuación con el siguiente ejemplo:

```
Ejemplo 1:
int  dim1[] = {5, 6, 7, 8, 9, 10};
int  dim2[] = dim1;
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados



6.1.2 Arreglo de objetos

Los arreglos de objetos verdaderamente como concepto serian como: un objeto que contiene más objetos, pero al tratarse de variables sería algo como una variable que guarda la referencia de las posiciones que guardan referencias del tipo declarado como objeto. Al igual que los arreglos primitivos se pueden declarar de igual forma, cuando se crea la instancia las posiciones de un arreglo son inicializadas por default en null.

Ejemplo 1:
```
String  []cadenas = new  String[6];
```

El siguiente ejemplo de un arreglo utilizando la notación corchetes.

Ejemplo 1:
```
String textos[]={"Literal","String","Literal", new String("String")};
```

Como comentario no olvidar que aquellas que fueron utilizadas con comillas dobles son reservadas en la sección de pool con la finalidad de reutilizarse los objetos.

6.2 Acceso a valores de un arreglo.

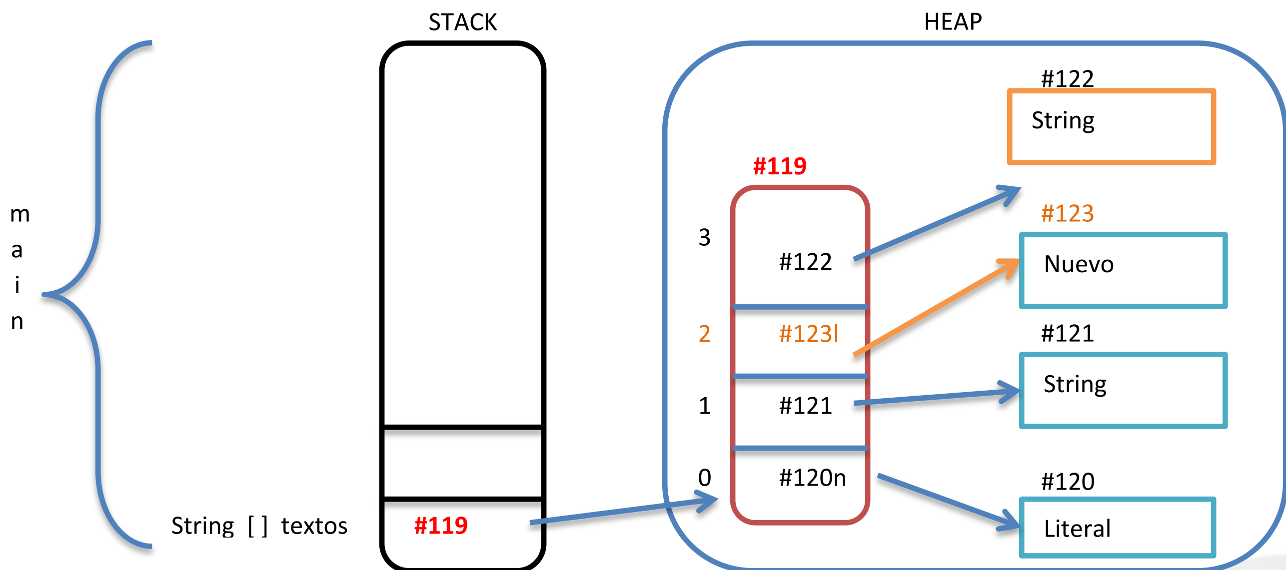
El acceso a esos valores de las posiciones se realiza mediante la notación corchete y una vez que se consigue el elemento se pude utilizar para asignarle un valor o para imprimirlo en pantalla, esto mismo aplica tanto a primitivos como de referencia,

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

ahora bien, si se accede a una posición invalida no será un error de compilación si no un error en tiempo de ejecución y lanza un *ArrayIndexOutOfBoundException*.

Ejemplo 1:
```
String textos[]={"Literal","String","Literal", new String("String")};
textos[2] = "Nuevo";
System.out.println("textos[2]: "+ textos[2]);//textos[2]: Nuevo
System.out.println("textos[4]: "+ textos[4]);// ArrayIndexOutOfBoundsException
```

Como se muestra en el ejemplo se realiza la modificación de la posición 2 y se obtiene para imprimirla en pantalla, como tipo es mucho más fácil obtener un resultado mediante un diagrama para ver cómo se comportan los elementos en heap-stack.



6.3 Uso del ciclo *for* mejorado.

Una de los retos que se tiene como arreglos en recorrerlos ya que están representadas mediante una variable, es por ello que los arreglos al ser objetos tienen un atributo que contiene la cantidad de elementos que al ser creados se le otorga, al cual este atributo es **lenght**.

Una forma de iterar esto es mediante el atributo mencionado el cual mediante la notación corchete se accede a la posición indicada ya sea para inicializar con valores en la iteración o para consultarlos sabiendo que es el tipo de dato declarado en el array, este atributo es utilizando con el bucle for el cual permite indicarle un inicio una condición de parada, es ideal cuando se trata de iterar rangos del arreglo.

Ejemplo 1:

```
String textos[]={"A","B","C","D","E","F"};

for(int i = 0; i < textos.lenght; i++ )
System.out.println(">"+ textos[i]);// Esto es un String y puede consultarse los métodos de un String
```

Ahora bien, se pude apreciar en el código anterior que es un poco tedioso manipular un arreglo utilizando la notación corchete por lo cual a partir de la versión de java 5 se incorporó el foreach el cual permite iterar toda una colección o un arreglo y la variable auxiliar contiene el valor de ese elemento actual en cada iteración.

A continuación, la segunda manera de iterar el mismo arreglo utilizando el for mejorado.

Ejemplo 1:

```
String textos[]={"A","B","C","D","E","F"};

for(String txt : textos)
System.out.println(">"+ txt); // Esto es un String y puede consultarse los métodos de un String
```

6.4 Uso de argumentos del método *main.*

Los argumentos del método main son principalmente para introducir parámetros de entrada a la aplicación, estos parámetros son coleccionados como un arreglo de cadenas.

La sintaxis básica para introducción de estos parámetros es dada por comando java seguido del nombre de la clase a ejecutar seguido de los argumentos separados por espacios, cabe mencionar que si se requiere una cadena identificado como un solo elemento del arreglo se encierran entre comillas dobles.

```
C:\ DirectorioProyecto >java NombreClase Argumento1 “Argumento 2” Arg3
```

Ahora bien, de entrada, los argumentos del método principal se encuentran inicializados por 0 elementos, por lo cual si se desea cambiar ese valor se necesita pasar de consola como se mostró en la sintaxis anterior.

Ahora bien, si se tiene el siguiente código y se ejecuta desde consola sin argumentos no tendremos ningún tipo de error, como se menciona el tamaño del arreglo por default del método main es de cero elementos y no necesariamente debe nombrarse como args, puede contener cualquier nombre de acuerdo a las reglas de sintaxis.

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

```java
// TestArgumentos.java

public class TestArgumentos{

   public static void main(String[] params) {
      System.out.println("Lenght: " + arg.lenght);
      for(String arg : params)
            System.out.println("Argumento: " + arg);
   }
}
```

Al ejecutar el código previo con la instrucción se tiene una salida similar como se muestra a continuación.

```
C:\ DirectorioProyecto >java TestArgumentos

Lenght: 0
```

Ahora si ese mismo código se ejecuta pasando algunos argumentos el resultado cambia como se muestra a continuación:

```
C:\ DirectorioProyecto >java TestArgumentos Arg1 “Argumento 2” Arg3 4

Lenght: 4
Argumento: Arg1
Argumento: Argumento 2
Argumento: Arg3
Argumento: 4
```

Es importante mencionar que, en el cuarto argumento, aunque se pase un 4 el método principal lo tomara como una cadena y no como un número.

6.5 Arreglos bidimensionales

Los arreglos bidimensionales son definidos como arreglos de arreglos, no necesariamente deben ser cuadráticas o matrices ya que cada posición puede tener tamaño diferente, pero del mismo tipo de dato ya sea primitivo o de referencia, la sintaxis para la declaración e inicialización no cambia.

Aquí algunos ejemplos de declaración valida de forma bidimensional en donde se aplican las mismas reglas cuando se requieran declarar más variables en la misma sentencia:

```java
Ejemplo:
String  []cads[],txts; //cads de 2 dimensiones, txts solo 1 dimensión.
int  [][]enteros,primos; //ambas variables son de 2 dimensiones.
char  caracteres[][],vocal; //caracteres es de 2 dimensiones, vocales solo es declarado como carácter.
```

La forma de inicialización no cambia mucho, pero si es algo importante mencionar, ya que por ejemplo en el caso de la inicialización por reservado de espacio de memoria, es obligatorio indicar solo en el primer corchete la cantidad de

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

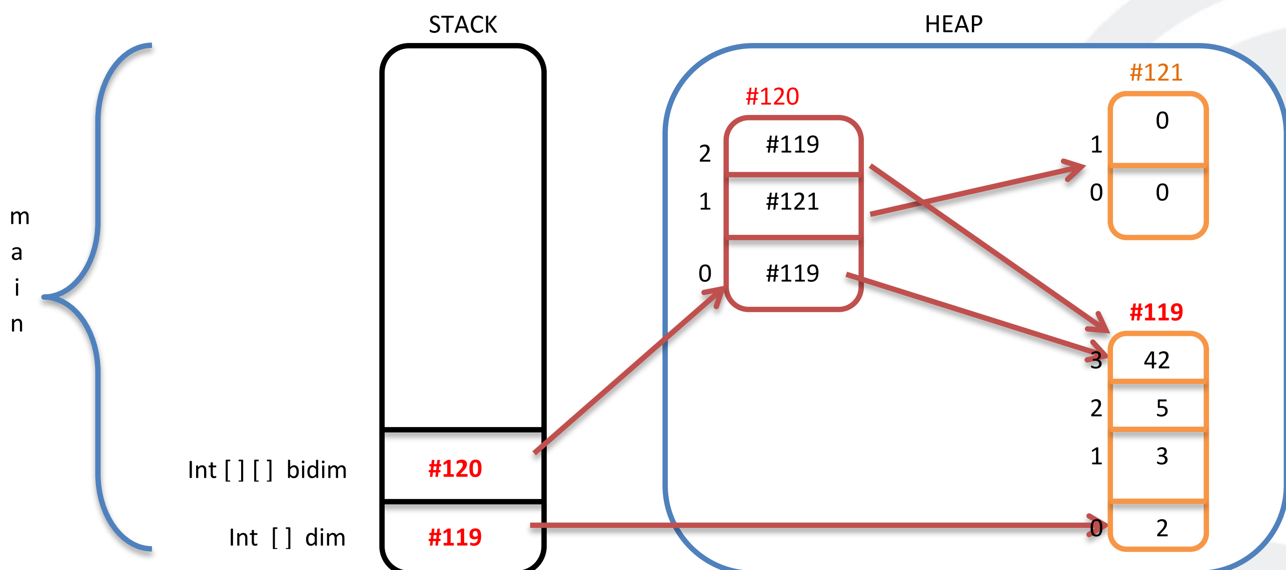
elementos y el segundo no necesariamente, a continuación, ejemplos validos de inicialización de los arreglos utilizando las 3 formas.

Ejemplo 1:
```
int [][]array = new int[5][]; //indica el primer arreglo tiene 5 posiciones inicializadas en null.
char cads[][] = new char [2][3]; //indica el primer arreglo tiene 2 posiciones y cada posición un arreglo de 3 posiciones.
String []msgs[]=new String[][]{"J","A"};//Al ser anónimo no lleva cantidad
Date mes[][] = { new Date(), null };// posición 0 una fecha y la siguiente null.
```

A continuación, un ejemplo de un arreglo bidimensional utilizando referencias.

Ejemplo:
```
int  dim1[] = { 2, 3, 5, 7 };
int bidim[][] = new int [3][];

bidim[0] = dim1;
bidim[1] = new int[2];
bidim[2] = dim1;

bidim[0][3] = 42;

for(int [] unidim : bidim) {
      System.out.println("\nLongitud: " + unidim.lenght);
      for(int elemento : unidim) {
            System.out.print(elemento + " ");
      }
}
```

Y bien, ahora ese ejemplo visto desde heap y stack.



Fundamentos de Programación en Java | Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

En base al diagrama anterior, la salida es muy similar a la que se muestra a continuación:

```
Longitud: 4
2 3 5 42
Longitud: 2
0 0
Longitud: 4
2 3 5 42
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

develop
talent & technology

MÓDULO 07

Métodos y constructores

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

7.1 Sintaxis y estructura de los métodos

Los métodos están dados para realizar un conjunto de actividades que estos a su vez pueden ser utilizados para realizar tareas complejas pertenecientes a la clase, aunque existen 3 tipos de métodos según el ámbito al que pertenece, pero todos ellos pueden recibir y retornar tipos de datos.

Estos métodos se usan en diferentes circunstancias, por el momento solo se mencionan la forma de declaración de estos métodos.

- **Métodos concretos:** son aquellos que le pertenecen a la instancia es como si fuera como un atributo y estos llevan cuerpo, estos le pertenecen a la instancia, son creados al momento de instanciar el objeto. La llamada a estos métodos se realiza mediante la instancia.

```
Sintaxis:
   [modificador_acceso] tipoDeRetorno nombreDeMetodo ([argumentos]) {
      //cuerpo del método
   }

Ejemplo:
public class Gerente {
   public void invocar() {
      System.out.println("Llamada al metodo");
   }

   public int consultarEdad() {
      return 0;
   }
}
```

- **Métodos de clase:** Son aquellos que le pertenecen a la clase y solo existe un método por la clase, esto quiere decir que, aunque crees varias instancias será compartido el mismo método. La diferencia que para su llamada se realiza mediante el nombre de la clase, estos son utilizados cuando la actividad es muy recurrente o no hay necesidad de instanciar si se sabe que solo se hace uso de un solo método o bien cuando se aplica una configuración global a un proyecto o en patrones de diseño como por ejemplo *Singleton,*; esta declaración lleva un modificador ***static*** como se muestra a continuación.

```
Sintaxis:
   [modificador_acceso] static tipoDeRetorno nombreDeMetodo([argumentos]) {
      //cuerpo del método
   }

Ejemplo:
public class Factory {
   public static Factory getFactory() {
     return /*implmentación de retorno*/
   }
}
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

- **Métodos abstractos:** Son aquellos que no tienen implementación, pero solo indican que hacer mediante la palabra reservada y modificador ***abstract***, mas no como debe implementarse, este tipo de métodos son utilizados solo para las clases abstractas e interfaces que solo indican que deben hacer, mas no como lo deben hacer, en el módulo de herencia se explica a fondo.

Sintaxis:
```
[modificador_acceso] abstract tipoDeRetorno nombreDeMetodo([argumentos]);
//Sin cuerpo del método
```

Ejemplo:
```
public abstract class Report{
   public abstract void generateReport(Banco banco);
}
```

7.2 Invocación de métodos de la misma clase

En este módulo se centra específicamente a los 2 primero tipos de métodos (concretos y de clase). Antes de hacer las llamadas a estos métodos es importante saber que antes de crear una instancia se carga la clase por la JVM, esto quiere decir que los métodos estáticos son cargados primero y sólo pueden ser utilizados por otros métodos estáticos, ahora bien, los métodos concretos que son creados a partir de una instancia y creados después de los de clase, éstos pueden utilizar tantos otros métodos concretos como métodos de clase hablando cronológicamente. Para hacer la llamada a métodos encontrándose en la misma clase y dependiendo el tipo de método se puede describir a continuación.

Ejemplo:
```
public class UtilileriasBanco {
      public void GenerateReport () {
            isBancoValido();
            UtilileriasBanco.getCountCuentaHabientes();
            this.imprimirCuentaHabientes();
      }

      public void imprimirCuentaHabientes() {

      }

      public static boolean isBancoValido() {
            return true;
      }

      public static int getCountCuentaHabientes() {
             isBancoValido();
             UtilileriasBanco.getCountCuentaHabientes();
      }

}
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

Como se puede observar cuando se hace uso de un método concreto se puede hacer la llamada mediante la palabra reservada **this**, el cual hace referencia a la instancia que lo que utilizando en ese momento o bien sin colocar la palabra **this,** estos métodos sí pueden usar métodos estáticos sin utilizar **this**. Mientras que en los métodos estáticos *getCountCuentaHabientes()* y *isBancoValido()* se hace la llamada sólo indicando el nombre del método desde un contexto estático, no se tiene permitido utilizar this en un contexto estático debido a que no fue declarado como un método concreto y además no se sabe de qué instancia es llamada, de lo contrario provocaría un error de compilación; también puede ser llamado con el nombre de la clase.

7.3 Invocación de métodos de diferente clase

La llamada a métodos desde una clase externo es muy frecuente ya que las utilerías se usan muy a menudo, es necesario que si la clase a utilizar se encuentra en un paquete diferente es necesario realizar la importación de la clase. He aquí un ejemplo de cómo se la clase UtileriasBanco se encuentra empaquetada en una clase.

```
package mx.com.develop.utilerias;

public class UtilileriasBanco {
      public void GenerateReport () {
            isBancoValido();
            UtilileriasBanco.getCountCuentaHabientes();
            this.imprimirCuentaHabientes();
      }

      public void imprimirCuentaHabientes() {    }

      public static boolean isBancoValido() { return true; }

      public static int getCountCuentaHabientes() {
             isBancoValido();
             UtilileriasBanco.getCountCuentaHabientes();
      }
}
```

Ahora bien, a continuación, se muestra el archivo fuente de la clase externa que utilizara los métodos, la diferencia es que ya se accede mediante la instancia aquellos métodos concretos de la instancia en específico, mientras que el método de clase se puede hacer mediante la instancia, aunque es mala práctica puesto si solo se hace uso de un método carga en memoria todo, o la manera de poder acceder es mediante el nombre de la clase, no olvidar la importación de esa clase externa.

```java
package mx.com.develop.test;
import  mx.com.develop.utilerias.UtileriasBanco;

public class TestBanco {
      public static void main(String [] args) {
            //Accediendo a métodos concretos mediante una instancia.
            UtilileriasBanco ub = new UtilileriasBanco();
            ub.imprimirCuentaHabientes();
            //Accediendo  metodos de clase
            UtilileriasBanco.isBancoValido();
      }
}
```

Existe otra manera para acceder a miembros estáticos pero no se recomienda como buena práctica ya que es no es conveniente cuando hay ambigüedad (duplicidad entre nombre) de variables, además de que no se sabe o se mal interpreta, esta forma es mediante una importación estática la cual permite importar tanto variables y métodos de clase aunque se puede especificar, es tal de colocar un asterisco se coloca la variable o método a utilizar y al utilizarlo solo se coloca el nombre de la variable o método estático.

```java
package mx.com.develop.test;
import  mx.com.develop.utilerias.UtileriasBanco;
import static mx.com.develop.utilerias.UtileriasBanco.*;

public class TestBanco {
      public static void main(String [] args) {
            //Accediendo a metodos concretos mediante una instancia.
            UtilileriasBanco ub = new UtilileriasBanco();
            ub.imprimirCuentaHabientes();

            //Accediendo  metodos de clase por medio de la importación static
            isBancoValido();
      }
}
```

7.4 Uso de argumentos

El uso de argumentos o parámetros son datos de entrada al método, esta información que se puede utilizar para realizar operaciones, procesarlas y posiblemente realizar un tipo de retorno y esta información a su vez puede ser pasada a otros métodos, cabe aclarar que cuando un valor es pasado como parámetro de un método verdaderamente se realiza una copia de ese valor ya sea de referencia o primitivo.

develop talent & technology

7.4.1 Declaración e invocación de métodos con argumentos

La sintaxis para la declaración de métodos con argumentos contenida dentro de los paréntesis de un método se realiza indicando el tipo de dato seguido de un nombre y se requiere más variables se separa por comas, cabe mencionar también que si existe ambigüedad de variables es decir duplicidad o confusión de variables locales o de instancia se hace uso de la palabra **this** para referirse a la variable de instancia.

Sintaxis:

```
[modificador_acceso] tipoDeRetorno nombreDeMetodo([type arg,type arg2]){}
```

Ejemplo:

```
public class Report{
      public Banco banco;
      public void generateReport(Banco banco,int cantidad){
            this.banco = banco;
            cantidad = 5;
      }
}
```

Al igual que la llamada a un método sin argumentos se realiza desde la misma manera, puede ser confuso cuando se trata de alterar la información de una referencia, puesto en los ejemplos se observa cómo se realiza algunas modificaciones y algunas otras no.

En el código se puede observar como al pasar parámetros cuando se tratan de valores primitivos el valor que recibe es copiado a la variable local o parámetro y no altera al valor original por lo cual imprime un 5.

Mientras que cuando se pasa una variable de referencia también es pasada la copia de la referencia de ese objeto, pero teniendo esa copia de esa referencia puedes alterar los atributos de esa referencia, por lo cual teniendo la referencia del gerente y mediante esa referencia puede cambiarse su nombre por Salvador, por lo cual al regresar el resultado será de Salvador.

Ahora bien, con la llamada al tercer método también se hace el paso de una variable de referencia pero la diferencia es que se está creando una nueva referencia pero solo a esa variable local, sin afectar a la original, dicho de otra forma se está creando una nueva referencia para la variable local mas no para la original y lo cual no daña al objeto original, por lo tanto la salida para la llamada al tercer método es también Salvador y no Pamela.

Ejemplo:

```
public class Gerente{
      public String name;
}

public class TestBanco {
      public void modificarValor(int valor) {
            ++valor;// equivalente a v = v+1;
      }

      public void modificarReferencia(Gerente gerente) {
            gerente.name = "Salvador";
      }

      public void modificarReferencia2(Gerente gerente) {
            gerente = new Gerente();
            gerente.name = "Pamela";
      }

      public static void main(String [] args) {
            TestBanco tb = new TestBanco();
            Gerente g = new Gerente();
            g.name = "Jose";
            int v = 5;

            //1. Paso por valor.
            tb.modificarValor(v);
            System.out.println("Valor: " + v);

            //2. Paso por referencia.
            tb.modificarReferencia(g);
            System.out.println("Nombre: " + g.name);

            //3. Paso por referencia.
            tb.modificarReferencia2(g);
            System.out.println("Nombre: " + g.name);
      }
}
```

7.5 Declaración de métodos con valor de retorno

Algo importante mencionar es que la tercera parte importante de un método es la salida o el tipo de retorno que pueda ocasionar un procesamiento al introducir parámetros.

Es importante mencionar que los métodos al regresar un tipo de retorno primitivo o de referencia obligatoriamente deben llevar la palabra reservada **return** dentro del cuerpo del método para finalizar sentencia y devolverlo inmediatamente.

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

Ejemplo:

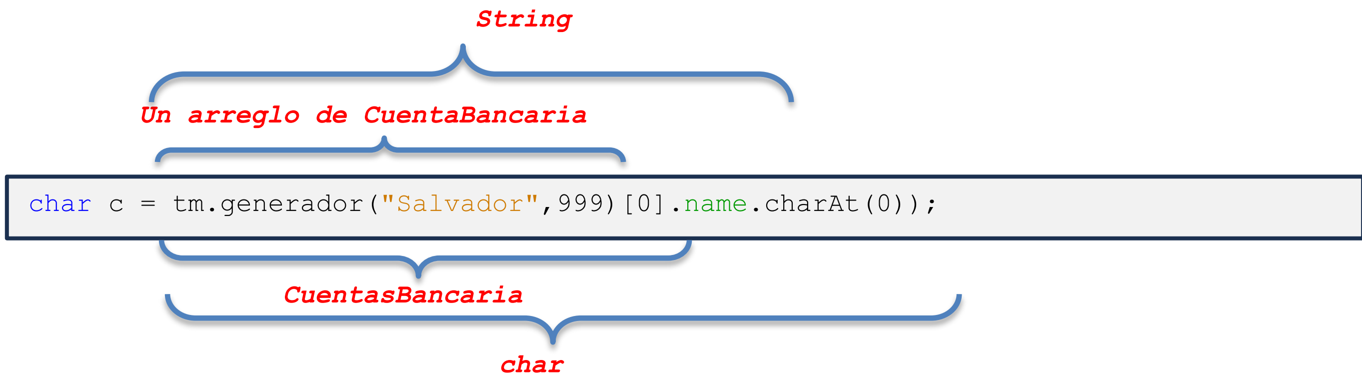
```
public class Gerente{
      int edad;
      String name;

      public String getName() {
            if(name == null){
                  return "Sin Nombre"
            }
            else{
                  return name;
            }
      }

      public static void main(String [] args) {
            Gerente g = new Gerente();
            System.out.println("Nombre: " + g.getName());
      }
}
```

La llamada a un método puede recibir diferentes tipos de datos y regresar un tipo de retorno diferente y este puede ser utilizado para asignarlo a una variable o bien para pasarlos como parámetro hacia otro método, como se ve en el ejemplo anterior al utilizarse en una concatenación del método *println*.

Una de las buenas prácticas que se tiene es que, si el tipo de retorno es usado para pasarlo hacia otro método, es recomendable asignarlo primero a una variable ya que si se realiza de esa manera puede ser más entendible. Veamos el siguiente ejemplo en donde procesa una serie de argumentos de diferente tipo, regresa un arreglo y se accede a una posición de ese arreglo que se accede a un atributo y a su vez se accede a un método del tipo de dato que se accede.

```
public class CuentaBancaria{
      int mount;
      String name;
}

public class TestMetodo{
      public static void main(String [] args) {
      TestMetodo tm = new TestMetodo();
      char c = tm.generador("Salvador",999)[0].name.charAt(0));
      System.out.println("Char: " + c);//Imprime la letra S
      }

      public CuentaBancaria[] generador(String name, int mount) {
            CuentaBancaria cb = new CuentaBancaria ();
            cb.name = name;
            cb. mount = edad;
            return new CuentaBancaria[]{cb};
      }
}
```

Para entender un poco de lo que verdaderamente se está realizando es recomendable entender hasta un segmento para saber con qué tipo de dato verdaderamente se está tratando veamos la forma de descomponer el dato, a continuación, una pequeña descripción de los tipos de datos que se están realizando.



7.6 Sobrecarga de métodos

La sobrecarga de métodos es indispensable cuando se quiere procesar información, pero se realiza con diferentes tipos de datos de entrada sin cambiar el nombre del método. Entre las reglas que destacan en la sobrecarga se encuentran:

- El nombre del método deberá llamarse igual.
- La lista de argumentos debe ser diferente ya sea en cantidad, orden o tipo de datos.
- El tipo de retorno puede ser diferente.
- El modificador de acceso puede ser diferente.

Bien he aquí un ejemplo de los métodos sobrecargados, e donde se aplica a la clase de utilerías que permite generar los reportes pero de acuerdo a diferentes tipos de listas ya sea de cuentahabientes o cuentas bancarias.

```
public class UtilileriasBanco {
      public static void GenerarReporte (CuentaHabiente[] clientes) {
            System.out.println("Reporte de clientes: ");
            for (CuentaHabiente cliente: clientes){
                  /*procesamiento*/
            }
      }

      public static void GenerarReporte (CuentaBancaria[] cuentas) {
            System.out.println("Reporte de cuentas: ");
            for (CuentaBancaria cuenta: cuentas){
                  /*procesamiento*/
            }
      }
}
```

Fundamentos de Programación en Java | Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

Ahora bien, los métodos se encuentran sobrecargados y además no existe confusión en el cual se va a mandar a llamar, debido al tipo de dato de arreglo que se pase será la invocación de ese método. Ahora bien, si se tratase de tipos de datos primitivos la prioridad se da de acuerdo a la tabla mencionada en el módulo 2 del tema promoción y casting.

Existe un problema al momento de utilizarlo el cual es la forma como se pasa el parámetro, para entender este punto se tiene la clase de prueba pasando el arreglo de CuentaHabientes sin problemas puesto recordar que los arreglos son objetos por lo cual verdaderamente se está pasando la copia de la referencia que apunta al arreglo.

```
public class TestReport {
      public static void main (String[] clientes) {
            //Arreglo con 4 cuentahabientes
            CuentaHabiente [] cuentaHabientes = {
                  new CuentaHabiente(),
                  new CuentaHabiente(),
                  new CuentaHabiente(),
                  new CuentaHabiente()
            };
            UtileriasBanco.generarReporte(cuentaHabientes);
      }
}
```

Ahora bien, si desea pasar los argumentos separados por comas marca un error de compilación, debido a que ahora no se pasa la referencia del arreglo si no los elementos separados por comas.

```
public class TestReport {
      public static void main (String[] clientes) {
            UtileriasBanco.generarReporte(new CuentaHabiente(),
                  new CuentaHabiente(),
                  new CuentaHabiente(),
                  new CuentaHabiente()
            );
      }
}
```

Para resolver este problema en java existe las variables argumentos que son tratados como arreglos en parámetros de un método y solo se pueden usar en parámetros de un método, que significa N cantidad de elementos a recibir abarcando desde 0 hasta N elementos, es decir de manera infinita, estas variables argumentos se representan con tres puntos suspensivos después del tipo de dato y antes del nombre de la variable.

```
Sintaxis:
    [modificador_acceso] tipoDeRetorno nombreDeMetodo(type ... name) {
       //cuerpo del método
    }
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

Para que compile correctamente la clase TestUtilerias se cambia a esta manera:

```
public class UtilileriasBanco {
	public static void GenerarReporte (CuentaHabiente... clientes) {
		System.out.println("Reporte de clientes: ");
		for (CuentaHabiente cliente: clientes){
			/*procesamiento*/
		}
	}

	public static void GenerarReporte (CuentaBancaria... cuentas) {
		System.out.println("Reporte de cuentas: ");
		for (CuentaBancaria cuenta: cuentas){
			/*procesamiento*/
		}
	}
}
```

Con esta modificación es permitida pasar ya sea una referencia del arreglo o bien elemento separados por comas, como se muestra a continuación y no marcara ningún error de compilación.

```
public class TestReport {
	public static void main (String[] clientes) {
		//Arreglo con 4 cuentahabientes
		CuentaHabiente [] cuentaHabientes = {
			new CuentaHabiente(),
			new CuentaHabiente(),
			new CuentaHabiente(),
			new CuentaHabiente()
		};
		UtileriasBanco.generarReporte(cuentaHabientes);

		//Pasando con 4 cuentahabientes separados por comas
		UtileriasBanco.generarReporte(new CuentaHabiente(),
			new CuentaHabiente(),
			new CuentaHabiente(),
			new CuentaHabiente()
		);
	}
}
```

Existen reglas al utilizar variables argumentos debido a que las variables argumentos tienden a ser infinitas, las cuales se describen a continuación.

- Están irán después de indicar los parámetros finitos, como último parámetro declarado.
- Solo puede tener un variable argumento declarada en el mismo método.

7.7 Modificadores de acceso

Los modificadores de acceso en Java son niveles de accesibilidad es decir que tanto es visible ya sea la clase, el método o el atributo en otras clases externas. Para entender este concepto la siguiente tabla muestra la visibilidad de cada nivel de acceso, cabe destacar que existen 4 niveles de acceso y 3 de ellos están representados por palabras reservadas salvo el que está por default.

Nivel de acceso	Descripción
Private	Visible solo para la clase donde fue declarado
Friendly-package (default)	Anterior + clases externas que se encuentren en el mismo paquete
protected	Anterior + clases que sean derivados de una clase base es decir por una clase generalizada. (herencia)
Public	Anterior + clases que se encuentren en paquetes diferentes.

Es importante también recordar que cuando se hace uso de un atributo se accede mediante una instancia y esa instancia está dada por una clase por lo cual es necesario realizar importación cuando se necesite. A continuación, un ejemplo del uso de 3 niveles de acceso en java (private, friendly package y public). El modificador protected se explica en el siguiente modulo.

```
package mx.com.develop.modelos;

public class CuentaHabiente {
      private String nombre;
      String direccion;
      public CuentaBancaria cuentaBancaria;

      public void manipular() {
            this.cuentaBancaria = new CuentaBancaria();
            this.direccion = "user;
            this.nombre = "user";

            CuentaHabiente ch = new CuentaHabiente();
            ch.cuentaBancaria = new CuentaBancaria();
            ch.direccion = "user;
            ch.nombre = "user";
      }
}
```

En esta primera clase, las 3 variables son utilizadas en la misma clase ya sea accediéndolas mediante *this* o mediante otra referencia creada encontrándose en la misma clase.

Ahora la siguiente clase que se encuentra en el mismo paquete solo podrán ser utilizadas las variables con modificador *friendly package* y *public* realizando la correspondiente importación siempre y cuando la clase sea pública y mediante una instancia de la referencia de la clase y no mediante *this*.

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

```java
package mx.com.develop.modelos;
/*no es necesaria la importacion ya que mabas clases se enceuntran en el mismo
paquete*/
public class Banco {

	public void conultar() {
		CuentaHabiente ch = new CuentaHabiente();
		ch.cuentaBancaria = new CuentaBancaria();
		ch.direccion = "user;
		ch.nombre = "user";
		//La compilacion de la linea de asignacion del nombre  falla debido a
		que es un atributo privado
	}
}
```

Ahora bien, si se tratase de una tercer clase pero esta se encuentra en un paquete diferente solo el atributo con modificador público es visible mediante la creación del objeto y la importación necesaria.

```java
package mx.com.develop.test;
import mx.com.develop.modelos.*;
public class TestBanco{

	public void manipular() {
		CuentaHabiente ch = new CuentaHabiente();
		ch.cuentaBancaria = new CuentaBancaria();
		ch.direccion = "user;
		ch.nombre = "user";
		//La compilacion de las ultimas dos asignaciones(direccion y
		nombre)falla debido a que es un atributo  con vicibilidad al apquete y
		privado correspondiente.
	}
}
```

7.8 Uso del encapsulamiento

El propósito del encapsulamiento se dice que es el ocultamiento de la información, bien el encapsulamiento es dado para ocultar la manera en que se implementa la lógica de cómo se asigna u obtiene un recurso el cual vienen siendo cajas negras en donde se conoce que hace, pero mas no la manera en como lo hace, esto es ideal cuando se tienen 2 contextos muy separados uno por la parte de donde se trabajan los modelos y su lógica y otra más por la parte donde se están utilizando esas interfaces.

Esto beneficia en muchas cosas por ejemplo en dar mantenimiento a las clases y agregar nuevas actividades al método encargado de realizar la o las tareas.

Veamos un ejemplo para tener más claro el concepto. Lo que el usuario al autentificarse hace es introducir sus datos para poder depositar desde un cajero automático y acceder a su cuenta, una vez que accede a su cuenta podrá hacer un deposito del dinero y actualizar la cuenta, pero la cuenta tiene un límite máximo que no debe pasar.

```
public class TestCajeroAutomatico {
      public static void main(String [] args) {
            CuentaHabiente cliente;
            cliente = Banco.conseguirCuentaHabiente("user","pass");

            CuentaBancaria cuenta = cliente.getCuentaBancaria();
            cuenta.saldo = cuenta.saldo + 1000;
      }
}
```

En el ejemplo anterior se muestra como al obtener tanto el cuentahabiente y la cuenta solo se interesa por saber que hacer mas no como lo hace debido a que al conseguir una cuentahabiente puede que se encuentre alojado en una base de datos, lo rescate de una archivo y/o se creen registros entre otras cosas, es por ello que es como una caja negra en donde a ti como usuario final te interesa saber qué hace mas no como lo hace, sin embargo cuando se está depositando el monto por ejemplo de 1000 en donde no se encuentra encapsulado, no sabemos cuánto se tenga en la cuenta pero si es un retiro que supere el monto esto altera el estado de la cuenta.

Es por ello que la encapsulación consiste en que los atributos deben colocarse con modificadores no necesariamente privados, pero tampoco públicos para restringir la manipulación directa de esos datos en clases externas y proporcionar interfaces llamadas setters para modificar el valor y getters para conseguir el valor.

7.8.1 Métodos get y set

La sintaxis básica de un ***setter*** consiste en un modificador de acceso público tipo de retorno vacío (void), el nombre del método llevara el prefijo set seguido del nombre del atributo al ser una palabra compuesta el nombre del atributo y cada separación de palabra inicia con mayúscula y recibe como parámetro una variable con tipo de dato igual que la declaración del atributo; su principal función es asignar el parámetro recibido a la variable de instancia.

Sintaxis:
```
public void setNombreAtributo(TipoAtributo nombre){nombreAtributo=nombre;}
```

La sintaxis básica de ***getter*** consiste en modificador de acceso público, tipo de retorno de igual declaración del atributo a regresar y en el nombre deberá llevar el prefijo get seguido del nombre del atributo, al ser una palabra compuesta el nombre del tributo y cada separación de palabra inicia con mayúscula y no recibe lista de argumentos; su principal función en regresar la variable de instancia.

Sintaxis:
```
public Tipo getNombreAtributo(){ return atributo;}
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

Bien una forma de crear la clase CeuntaHabiente y CuentaBancaria de manera encapsulada seria restringir el acceso de los atributos colocando el modificador private y generar set y get del atributo.

Las siguientes figuras muestra como las clases cuentahabiente y cuenta bancaria se encuentran de manera encapsulada y con ello es más fácil restringir o tomar medidas encargado de modificar el valor u obtenerlo. Ahora bien es aquí que cuando la variable local al llamarse igual que el atributo existe ambigüedad el método (duplicidad de nombres) es por ello que se opta por colocar la palabra reservada **this** para referirse a la instancia y poder acceder a su atributo y aquella que no lleva this hace referencia a la variable local, se observa también que en los **get** no es necesario colocar this debido a que en ese método no existe ambigüedad.

```
public class CuentaHabiente {

	private String nombre;
	private String direccion;
	private CuentaBancaria cuentaBancaria;

	public String getNombre() {
	    return nombre;
	}

	public void setNombre(String nombre) {
	    this.nombre = nombre;
	}

	public String getDireccion() {
	    return direccion;
	}

	public void setDireccion(String direccion) {
	    this.direccion = direccion;
	}

	public CuentaBancaria getCuentaBancaria() {
	    return cuentaBancaria;
	}

	public void setCuentaBancaria(CuentaBancaria cuentaBancaria) {
	    this.cuentaBancaria = cuentaBancaria;
	}
}
```

Algo importante que cabe resaltar en la encapsulación es cuando se trata de constantes debido a como son contantes están no tienen permitido modificarse por lo cual se puede solo tener el método getter.

```java
public class CuentaBancaria {
    private double final LIMITE_MAX = 450_000;
    private String numeroDeCuenta;
    private double saldo;

    public String getNumeroDeCuenta() {
        return numeroDeCuenta;
    }

    public void setNumeroDeCuenta(String numeroDeCuenta) {
        this.numeroDeCuenta = numeroDeCuenta;
    }

    public double getSaldo() {
        return saldo;
    }

    public void setSaldo(double saldo) {
         int montoTemp = this.saldo + saldo;
         if(montoTemp <= LIMITE_MAX){
            this.saldo = montoTemp;
         }
         else{
            System.out.println("Ha superado el maximo permitido,");
            System.out.println("Consulte a su Gerente para incrementar");
            System.out.println("El dinero se depositado se le regresara.");
         }
    }

    public double getLIMITE_MAX () {
        return LIMITE_MAX;
    }
}
```

Como se muestra al modificar un saldo trabajo como un método de actualizado, en donde toda la lógica está contenida en un método. A este tipo de restricciones se le conoce como reglas de negocio que no son más que otra cosa que restricciones que tiene el sistema para que trabaje correctamente. Ahora sí, con base en esto la clase de prueba podrá utilizarla sin problema y todas las restricciones estarán cubiertas por el método encargado modificar el salario.

```java
package mx.com.develop.test;

public class TestCajeroAutomatico {
      public static void main(String [] args) {
            CuentaHabiente cliente;
            cliente = Banco.conseguirCuentaHabiente("user","pass");

            CuentaBancaria cuenta = cliente.getCuentaBancaria();
            Cuenta.setSalario(1000);
      }
}
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

7.9 Creación de constructores

Los constructores son como métodos salvo que estos no llevan tipo de retorno y el nombre es el mismo que la clase y además pueden recibir argumentos y pueden tener 4 posibles niveles de acceso. Su principal función es inicializar variables cuando se crea el objeto o realizar tareas al momento de instanciarse.

```
Sintaxis:
public NameClass([Arguemntos]){}
```

7.9.1 El constructor default

Todas las clases de java tienen un constructor por default, es por ello que cuando se realiza por ejemplo la instrucción new CuentaHabiente() y no se tenía la declaración de un constructor el código compilaba perfectamente, lo que verdaderamente se realiza es la ejecución de ese constructor que esta por default para inicializar, aunque no se vea de forma explícita existe, como se muestra continuación el ejemplo de la clase test usando un constructor que esta por default.

```
public class TestCajeroAutomatico {
      public static void main(String [] args) {
            CuentaHabiente cliente = new CuentaHabiente();
      }
}
```

En la siguiente se muestra como se encuentra el constructor por default de forma comentada.

```
public class CuentaHabiente {
/***        ATRIBUTOS   ***/
      private String nombre;
      private String direccion;


      /*CONSTRUCTOR IMPLICITO
      public CuentaHabiente (){   }*/


/***        MÉTODOS     ***/
      public String getNombre() {
          return nombre;
      }
      public void setNombre(String nombre) {
          this.nombre = nombre;
      }
      public String getDireccion() {
          return direccion;
      }
      public void setDireccion(String direccion) {
          this.direccion = direccion;
      }
}
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

Cuando nuestra clase se le coloca un constructor de manera explícita sin argumentos o recibiendo argumentos, el constructor por default desaparece y el desarrollador está obligado a utilizar el constructor el cual se definió de forma explícita con los argumentos que se establecieron, de lo contrario marca un error de compilación, como buena práctica si se colocan constructores de forma explícita es recomendable colocar entre la declaración de los atributos y los métodos.

A continuación, se muestra el uso del constructor de manera explícita.

```
public class TestCajeroAutomatico {
      public static void main(String [] args) {
         CuentaHabiente cliente = new CuentaHabiente("Salvador","Roma 46");
      }
}
```

Y bien, el modelo declarando el constructor de forma explícita.

```
public class CuentaHabiente {
/***        ATRIBUTOS   ***/
      private String nombre;
      private String direccion;

/***        CONSTRUCTORES     ***/
      public CuentaHabiente (String nombre, String direccion ){
            this.nombre = nombre;
            this.direccion = direccion;
      }

/***        METODOS     ***/
      public String getNombre() {
          return nombre;
      }

      public void setNombre(String nombre) {
          this.nombre = nombre;
      }

      public String getDireccion() {
          return direccion;
      }

      public void setDireccion(String direccion) {
          this.direccion = direccion;
      }
}
```

7.9.2 Sobrecarga de constructores

Al igual que los métodos existen constructores sobrecargados que permiten maneras alternas de instanciar un objeto, la única diferencia con los métodos y constructores es que estos últimos no llevan tipo de retorno y los argumentos deben ser diferentes ya sea en cantidad, tipo u orden y el modificador de acceso también puede ser diferente.

La llamada a otros constructores de la misma clase es posible en siempre y cuando se encuentre en alguno de los constructores de la misma clase y debe realizarse como primera sentencia del cuerpo del constructor mediante la palabra **this** seguido de un par de paréntesis en donde se pasa de la lista de argumentos del constructor que se requiera llamar.

A continuación, un ejemplo de la manera de cómo poder utilizar constructores sobrecargados

```
public class CuentaHabiente {
      private String nombre;
      private String direccion;

      public CuentaHabiente (String nombre, String direccion ){
            this.nombre = nombre;
            this.direccion = direccion;
      }

      public CuentaHabiente (){
            this("Sin Nombre","Sin direccion");
      }

      public String getNombre() {
          return nombre;
      }

      public void setNombre(String nombre) {
          this.nombre = nombre;
      }

      public String getDireccion() {
          return direccion;
      }

      public void setDireccion(String direccion) {
          this.direccion = direccion;
      }

      public void getInformation(){
            System.out.println("Nombre: " + name);
            System.out.println("Direccion: " + this.direccion +"\n");

      }
}
```

En el ejemplo anterior se puede observar que si se crea un cuentahabiente con el constructor sin argumentos su primera línea de código manda a llamar al constructor que recibe dos strings y estos al pasarse como parámetros a ese constructor se ejecuta y asigna por default esos valores, es conveniente cuando en tal de que tenga el valor por default en null mande datos coherentes. En cambio, sí se crea una instancia utilizando el constructor con ambos parámetros estos si se les asignara a los de instancia, a continuación un ejemplo de cómo se utilizan las dos situaciones.

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

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

      CuentaHabiente user1 = new CuentaHabiente("Salvador","Roma 46");
      user1.getInformation();

      CuentaHabiente user2 = new CuentaHabiente();
      user2.getInformation();
   }
}
```

Al ejecutar el código se tiene una salida muy coherente con datos más entendibles.

```
Nombre: Salvador
Direccion: Roma 46

Nombre: Sin Nombre
Direccion: Sin Direccion
```

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

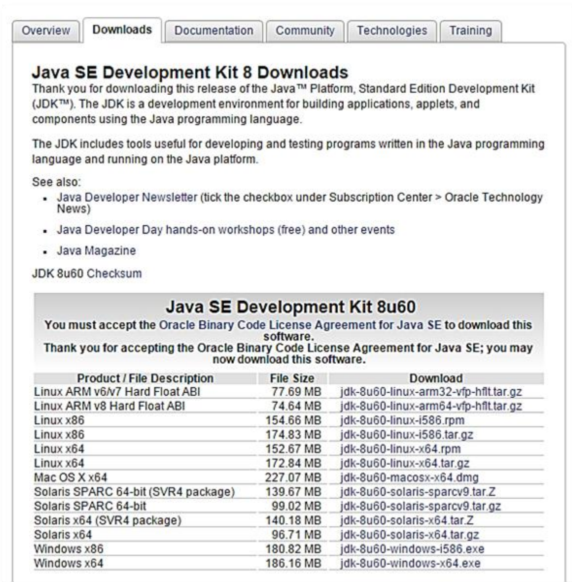
develop
talent & technology

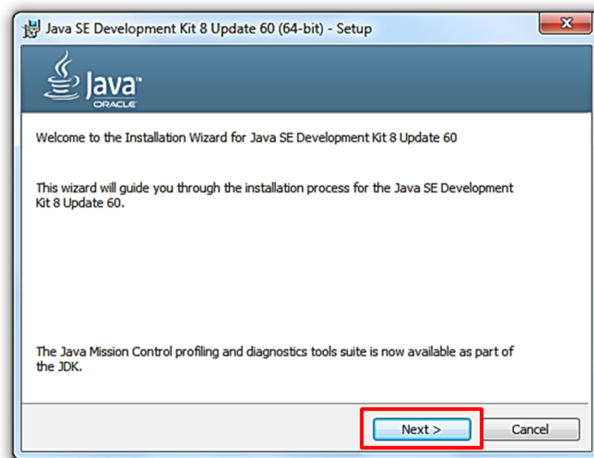
Apéndice

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados

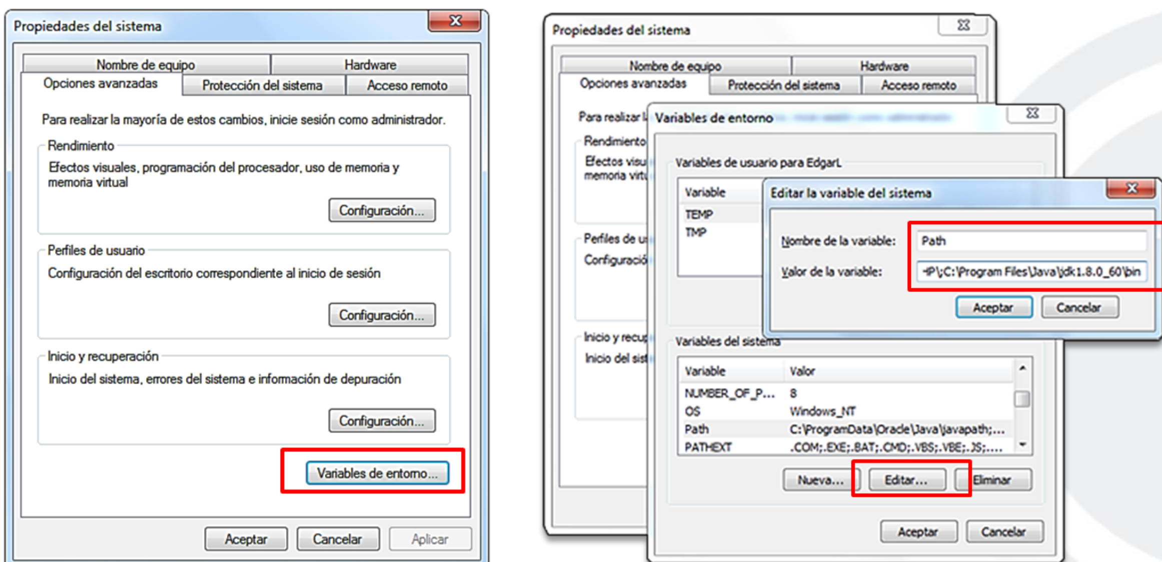
Instalación y Configuración del JDK

El JDK puede ser descargado desde la página oficial de Oracle®, donde se puede obtener la versión del instalador dependiendo del Sistema Operativo donde va a ser utilizado. En el caso de Windows®, el archivo se encuentra empaquetado con la extensión .exe y que al ejecutarlo desplegará un Wizard que guiará durante todo el proceso de instalación.





Una vez instalado el JDK es necesario configurarlo para utilizar sus herramientas desde línea de comandos, para esto se requiere asignar una variable de entorno que indica al sistema operativo en dónde se encuentran ubicados los programas que sean mandados a llamar con instrucciones en consola.



Como muestra la imagen, en las propiedades del sistema de Windows se encuentra un botón llamado "Variables de entorno", al dar clic se despliega una ventana para editar o agregar nuevas variables del sistema. Se debe seleccionar una

llamada Path y hacer clic en “editar” para desplegar una pequeña ventana donde se realiza la configuración. El valor de la variable contiene una cadena de texto con información de otros programas, al final de esa cadena se agrega un punto y coma (;), sin dejar espacio se indica la ubicación del directorio bin del JDK y se da clic en el botón “aceptar”.

Después de haber instalado y configurado el Kit de Desarrollo (JDK), es posible comenzar a escribir el código de las aplicaciones en un procesador de texto para después compilar y ejecutar desde línea de comandos. También se pueden utilizar programas que ayudan al programador con diversas tareas como: ingresar código a través de wizards, detectar los errores en la sintaxis del lenguaje, agilizar la escritura de instrucciones mediante shortcuts, crear los paquetes de clase, etcétera. Estos programas son llamados Entornos de Desarrollo Integrado (Integrated Development Environment, IDE), los más utilizados son NetBeans, Eclipse y JDeveloper.

Nota: La variable Path contiene información crítica de otros programas y del propio Sistema Operativo, es importante no borrar nada en su cadena de valor. Únicamente se debe agregar un *punto y coma* como separador y especificar la ruta del directorio que contiene las herramientas del JDK.

Copyright 2016 © Develop Talent & Technology. Todos los derechos reservados