

MANUAL DE FUANDAMENTOS DE JAVA

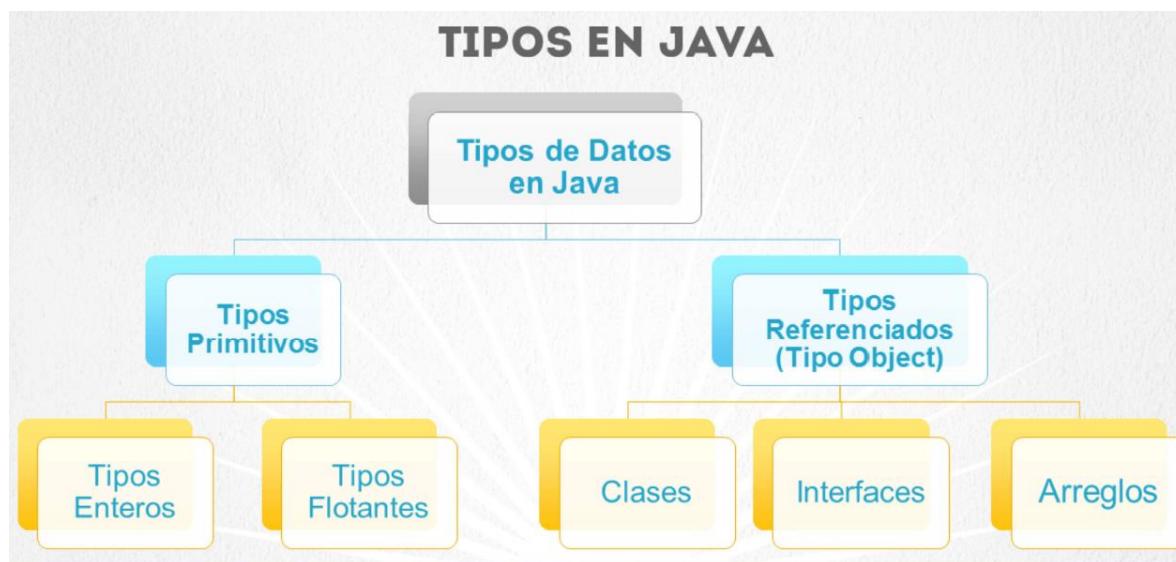
INDICE

1.- VARIABLES EN JAVA.....	3
2.-TIPOS PRIMITIVOS EN JAVA.....	5
3.- OPERADORES.....	6
4.- SENTENCIAS DE CONTROL.....	7
5.- CICLOS EN JAVA	8
6.- CREACIÓN DE CLASE EN JAVA.....	10
7.- OBJETOS EN JAVA	13
8.- MÉDOTOS EN JAVA	14
9.- CONSTRUCTORES EN JAVA.....	16
10.- ALCANCE DE VARIABLES	17
11.- MEMORIA STACK Y HEAP EN JAVA.....	19
12.- PASO POR VALOR Y PASO POR REFERENCIA.....	21
13.- REGRESO DE UN MÉTODO Y PALABRA RETURN	23
14.- PALABRA this EN JAVA	24
15.- PALABRA null EN JAVA	26
16.- ENCAPSULAMIENTO EN JAVA.....	28
17.- CONTEXTO ESTÁTICO	32
18.- HERENCIA EN JAVA	36
19.- SOBRECARGA DE CONSTRUCTORES.....	41
20.- SOBRE CARGA DE MÉTODOS.....	45
21.- PAQUETES JAVA.....	48
22.- PALABRA FINAL EN JAVA	51
23.- ARREGLOS EN JAVA	53
24.- MATRICES EN JAVA.....	59
25.- DISEÑO DE CLASE EN JAVA	64

1.- VARIABLES EN JAVA

El objetivo de declarar una variable es reservar espacio de memoria dependiendo del tipo que vayamos a utilizar.

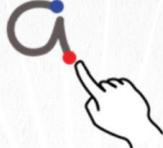
Estas variables nos permiten también hacer programas dinámicos, por lo que en la mayoría de los casos los valores cambiarán durante la interacción con el usuario y el programa.



CADERAS EN JAVA

TRATAMIENTO ESPECIAL

NO ES UN TIPO PRIMITIVO



CONTEXTO STRING

NO SE NECESITA INSTANCIAR

Ej. String saludo = "Hola Mundo";
 Comparación: saludo.equals("Hola Mundo");

CARACTERES DE ESCAPE AL UTILIZAR CADENAS

Secuencia de Escape	Descripción
\t	Inserta un tabulador
\b	Inserta un retroceso (backspace)
\n	Inserta una nueva línea
\r	Inserta un retorno de carro
\f	Se mueve a la siguiente pagina (Form feed). Se utiliza para impresoras, no en consolas.
'	Inserta una comilla simple
"	Inserta una comilla doble
\	Inserta una barra invertida

CLASE SCANNER

```
import java.util.Scanner; //LA LIBRERIA

public class Main { // NONRE D ELA CLASE
    public static void main(String[] args) {//METODO
        Scanner scanner = new Scanner(System.in);
        //1      //2      //3   //4      //5
        //1 NOMBRE DEL METODO
        //2 NOMBRE DE LA VARIABLE
        //3 LLAMANDO EL METODO
        //4 LLAMANDO EL METODO
        //5 LEER CONSOLA DE ENTRADA
        System.out.println("Proporciona tu nombre:");
        var usuario = scanner.nextLine();
        var saludar = "Saludos";
        System.out.println(saludar + " " + usuario);
    }
}
```

2.-TIPOS PRIMITIVOS EN JAVA

CONVERSIÓN DE TIPOS PRIMITIVOS EN JAVA

- **Convertir una cadena a un tipo entero**

```
Int o var edad = Integer.parseInt(s:"20");
System.out.println("edad = " +edad);
```

- **Convertir una cadena a tipo double**

```
double valorPi = Double.parseDouble("3.14");

System.out.println("valorPi = " + valorPi);
```

- **Convertir cadena a char**

```
char c = "hola".charAt(0);
System.out.println("c = " + c);
```

- **Convertir la variable scanner a entero, flotante, double, y booleano**

```
var o Scanner scanner = new Scanner(System.in);
int entero = Integer.parseInt(scanner.nextLine());
float floatente = Float.parseFloat(scanner.nextLine());
double double1 =Double.parseDouble(scanner.nextLine());
```

- **Convertir una variable Scanner a char**

```
var o Scanner scanner = new Scanner(System.in);
char caracter = scanner.nextLine().charAt(0);
System.out.println("caracter = " + caracter);
```

- **Convertir tipo entero,decimal y booleanos a cadena**

```
String valorTexto = String.valueOf(3);
System.out.println("valorTexto = " + valorPiTexto);
```

- **Otra manera convertir entero, decimal y booleanos a cadena**

```
String valorPiTexto2 = "" + valorPi;
System.out.println("valorPiTexto2 = " + valorPiTexto2);
```

- **Convertir bytes a short**

```
byte b = 10;
short s = b;
```

- **Convertir short a byte**

```
short s2 = 15;
byte b2 = (byte) (s2 + 1);
```

3.- OPERADORES

OPERADORES	
Operadores	Operador
Operadores aritméticos	+ , - , * , / , %
Operadores de relación	< , > , <= , >= , != , ==
Operadores lógicos	&& o & , o , ! , ^
Operadores unitarios	~ , -
Operadores a nivel de bits	& , , ^ , << , >> , >>>
Operadores de asignación	++ , -- , = , *= , /= , %= , += , -= , <<= , >>= , >>>= , &= , = , ^=
Operador condicional	?:
Prioridad y orden de evaluación	() , [] , . , - ~ ! ++ -- , new (tipo) expresión , */ %, + - , <<>> , >>> , < <= > >= , == != , & , ^ , , && , , ?:

```

System.out.println("f = " + f);//3

import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        int a = 3, b = 2;

        // COMPOSICION EN JAVA
        a += 1;//a=a+1
        System.out.println("a = " + a);

        // OPERADORES UNARIO
        a = 3;
        b = -a;
        System.out.println("b = " + b); // -3

        //OPERADOR TIPO BOOLEAN
        boolean c = true;
        boolean d = !c;

        System.out.println("d = " + d); //false

        //OPERADOR POSINCREMENTO
        Int e = 3;
        Int f = e++;
        System.out.println("e = " + e); //4
    }
}

//OPERADOR PREINCREMENTO
int g = 3;
int h = ++g;
System.out.println("g = " + g);//4
System.out.println("h = " + h);//4

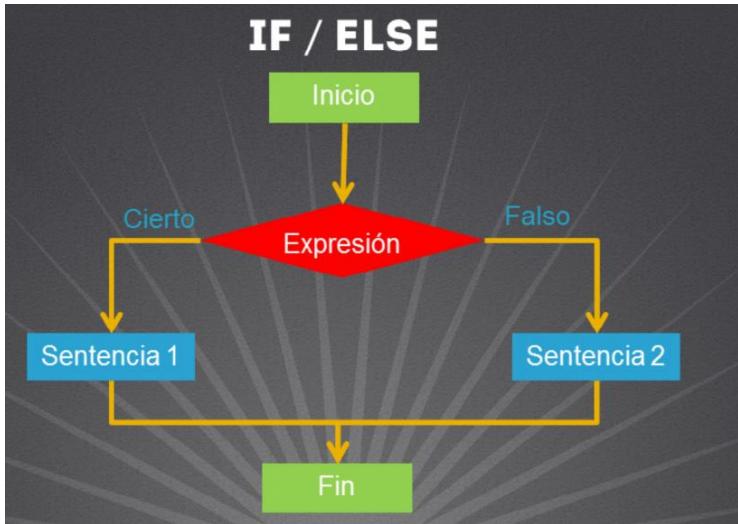
//OPERADORES RELACIONALES
a = 3;
b = 2;
boolean i = a == b;
System.out.println("i = " + i); //false

//PARA REALIZAR COMPARACIONES
DE TEXTO SE UTILIZA EQUALS
EJEMPLO
String j = "holá", k = "adiós";
boolean l = j.equals(k);

b -= 1;//b = b - 1
System.out.println("b = " + b);
// +=, /=, %=

```

4.- SENTENCIAS DE CONTROL



SINTAXIS

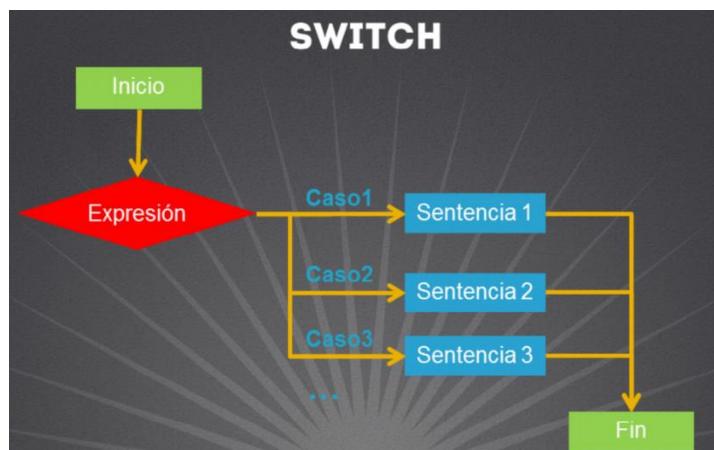
if(condicion)

//Sólo una sentencia si no se usan corchetes

else

//Sólo una sentencia si no se usan corchetes

if(condicion) //Sólo una sentencia si no se usan corchetes else //Sólo una sentencia si no se usan corchetes



SINTAXIS SWITCH

Valores *byte, short, int, char o String*

```
switch (expresión) {  
    case valor1:  
        //Sentencias  
        break;  
    case valor2:  
        //Setencias  
        break;  
    case valorN:  
        //Setencias  
        break;  
    default:  
        //Setencias  
}
```

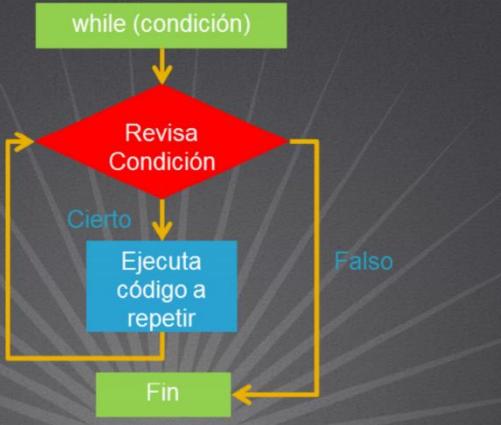
Cada case termina con dos puntos

Las sentencias si llevan punto y coma

el *default* es opcional

5.- CICLOS EN JAVA

DIAGRAMA FLUJO CICLO WHILE



SINTAXIS CICLO WHILE

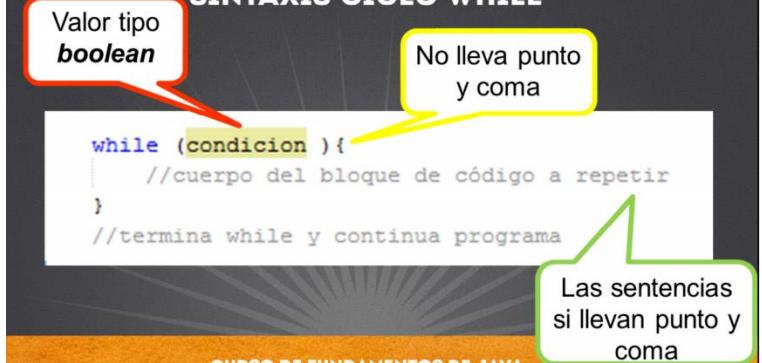
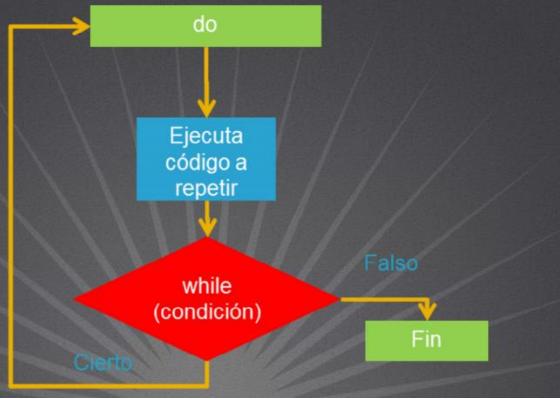


DIAGRAMA FLUJO CICLO DO WHILE



SINTAXIS CICLO DO WHILE

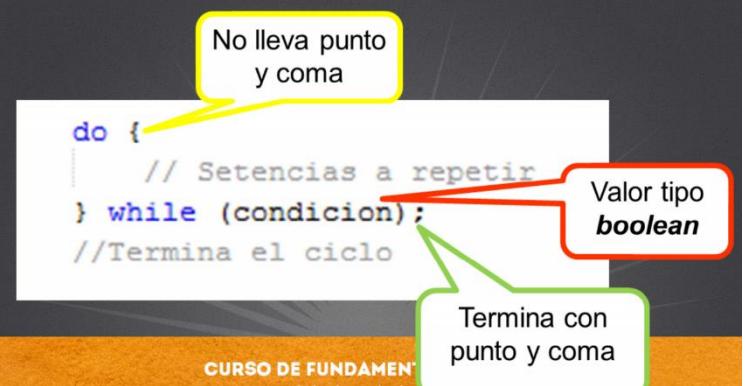
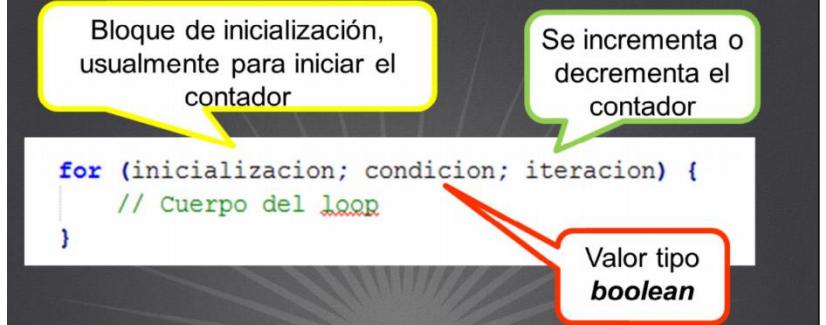


DIAGRAMA FLUJO CICLO FOR



SINTAXIS CICLO FOR



Sentencias continue y break

1. Break : es para romper el ciclo del programa ejemplo

```
For(int i = 0 ; i < 3 ; i++) {  
    If (i % 2 ==0)  
        System.out.println(" I = " +i);  
    Break;  
}  
Imprime solamente  
I =0 porque romper el ciclo
```

2. Continue es para continuar el ciclo del programa

```
For( int i = 0; i < 3; i++){  
    If ( i % 2 !=0)  
        Continue;  
    System.out.println(" i =" +i);  
}
```

Imprime solamente I

I = 0
I = 2

6.- CREACIÓN DE CLASE EN JAVA

Clase



- Define la naturaleza de un objeto
- Es una plantilla que puedes crear objetos
- Es instancia de un objeto
- Posee atributos y métodos
- Posee un nombre
- Molde



FORMA GENERAL DE UNA CLASE EN JAVA

Nombre de la clase debe terminar con extensión .java:

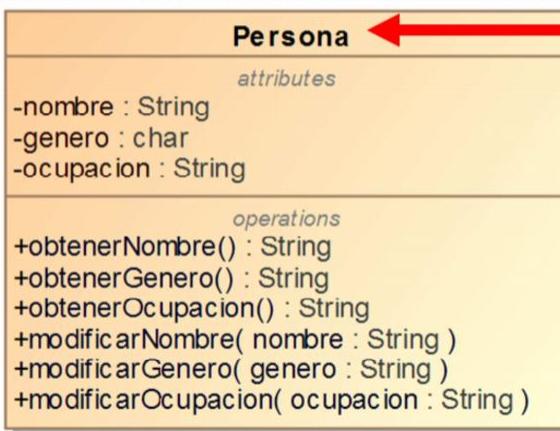
```
class NombreDeLaClase {  
  
    tipoDato variableDeInstancia1;  
    tipoDato variableDeInstancia2;  
    //Más variables de instancia...  
  
    tipoDato nombreDelMetodo1(argumentos) {  
        //Cuerpo del método  
    }  
  
    tipoDato nombreDelMetodo1(argumentos) {  
        //Cuerpo del método  
    }  
    //Más métodos...  
}
```

FORMA GENERAL DE UNA CLASE EN JAVA

Nombre de la clase debe terminar con extensión .java:

```
class NombreDeLaClase {  
  
    tipoDato variableDeInstancia1;  
    tipoDato variableDeInstancia2;  
    //Más variables de instancia...  
  
    tipoDato nombreDelMetodo1(argumentos) {  
        //Cuerpo del método  
    }  
  
    tipoDato nombreDelMetodo1(argumentos) {  
        //Cuerpo del método  
    }  
    //Más métodos...  
}
```

DIAGRAMA GENERAL DE UNA CLASE EN JAVA



Nombre de la Clase

Atributos

Métodos

Manera de crear un método no regrese ningún tipo de valor

```
Public void despegarNombre () {  
}
```

7.- OBJETOS EN JAVA

Objeto



- Es la instancia de una clase
- Puede cambiar los valores
- Llamar un método
- Los objetos contiene características ejemplo clase persona sus características son nombre, color, genero, escolaridad, etc.

CREACION DE UN OBJETOS EN JAVA

Creación de un objeto Persona:

```
public class PersonaPrueba {  
  
    public static void main(String args[]) {  
        //Creacion de un objeto  
        Persona p1 = new Persona();  
  
        //Modificar valores  
        p1.nombre = "Armando";  
        p1.apellidoPaterno = "Esparza";  
        p1.apellidoMaterno = "Lara";  
    }  
}
```

Para definir un objeto vamos a utilizar la palabra reservada `new`. Esta palabra en Java significa en términos simples que vamos a crear una nueva variable del tipo Persona. Para crear un objeto vamos a utilizar una línea de código como sigue:

`Persona p1 = new Persona();`

Para acceder a los atributos de nuestras clases vamos a utilizar el operador `.` A través de este operador se liga el nombre del objeto con el nombre del atributo de la clase. Por ejemplo, para asignar el valor de Armando al atributo nombre del objeto `p1`, sería como sigue:

`p1.nombre = "Armando"`

8.- MÉTODOS EN JAVA

MÉTODOS EN JAVA

Definición general de un método en Java:

```
class NombreClase{  
  
    tipo nombreMetodo(lista_de_argumentos){  
        //Cuerpo del método  
    }  
}
```

Ejemplo de un método en Java:

```
public class Aritmetica{  
  
    int sumar(int a, int b){  
        //realiza la suma y regresa el resultado como un entero  
        return a + b;  
    }  
}
```

También es común en Java utilizar la notación de camello para escribir los nombres tanto de variables como de métodos, y en general de los nombres que escribimos. Esta notación consiste en escribir la primera letra en minúscula y posteriormente la primera letra de cada palabra debe ir en mayúscula.

LLAMANDO A UN MÉTODO

Llamada general de un método en Java:

```
//Crear un objeto de la clase a llamar su método  
TipoClase objeto = new TipoClase();  
  
//Llamamos el método, enviando argumentos si se requieren  
//Si el método regresa un valor podemos recibirla según el tipo  
tipoDevuelto resultado = objeto.nombreMetodo(arg1, arg2, etc);
```

Ejemplo de llamada a un método en Java:

```
//Creamos un objeto de la clase Aritmetica  
Aritmetica a = new Aritmetica();  
  
//Llamamos el metodo sumar y recibimos el valor devuelto  
int resultado = a.sumar(5, 3);
```

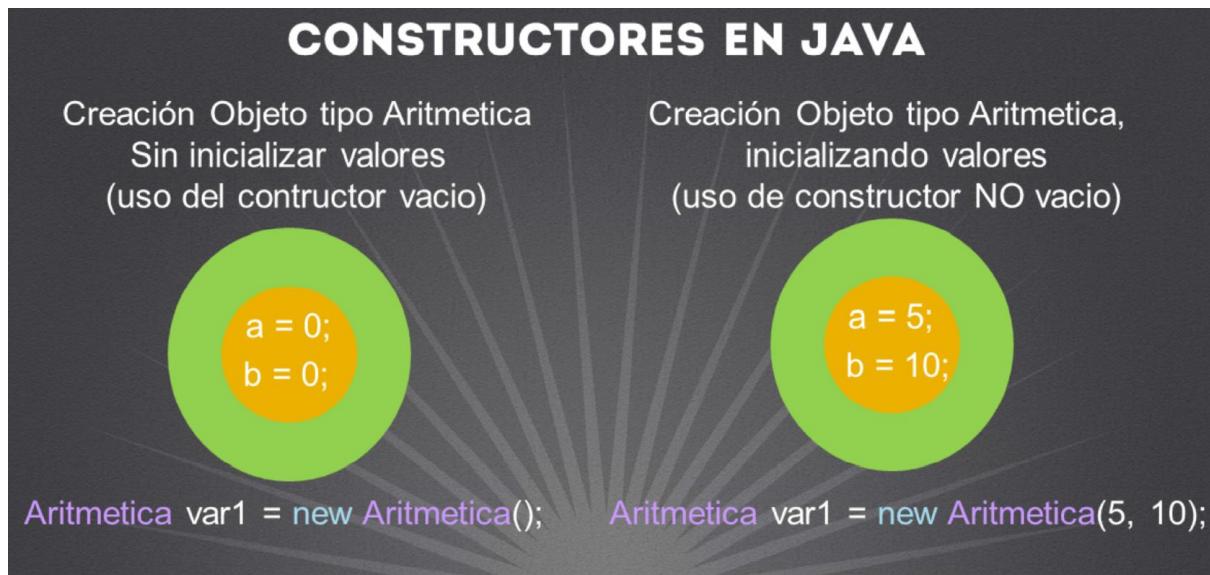
Una vez que hemos agregado un método a una clase, podemos hacer uso de él haciendo lo siguiente:

- 1) Debemos declarar un objeto del tipo de la clase que tiene el método que nos interesa utilizar.

- 2) Por medio del objeto declarado, utilizamos el operador punto (.) y posterior a este operador escribimos el nombre del método que nos interesa llamar. Recordemos que en Java, los nombres son sensibles a mayúsculas y minúsculas, por lo que debemos de hacer la llamada al método tal y como lo hayamos escrito, respetando las mayúsculas y minúsculas que hayamos utilizado para definir el método.
- 3) Posterior del nombre del método, abrimos y cerramos paréntesis. Si el método no recibe ningún argumento, solo abrimos y cerramos paréntesis, sin embargo si el método fue escrito para recibir argumentos, debemos proporcionar los argumentos del tipo esperado, separando por comas cada uno de ellos.
- 4) Finalmente, si el método fue escrito para regresar un valor, es opcional el recibirla con una variable del tipo que regresa el método. Debemos notar que el tipo devuelto por un método no tiene que ser igual a los tipos de los argumentos recibidos. Para ello debemos observar la firma del método que vayamos a utilizar. Si recordamos, la firma del método sumar sería:

```
int sumar(int, int); //firma del método
```

9.- CONSTRUCTORES EN JAVA

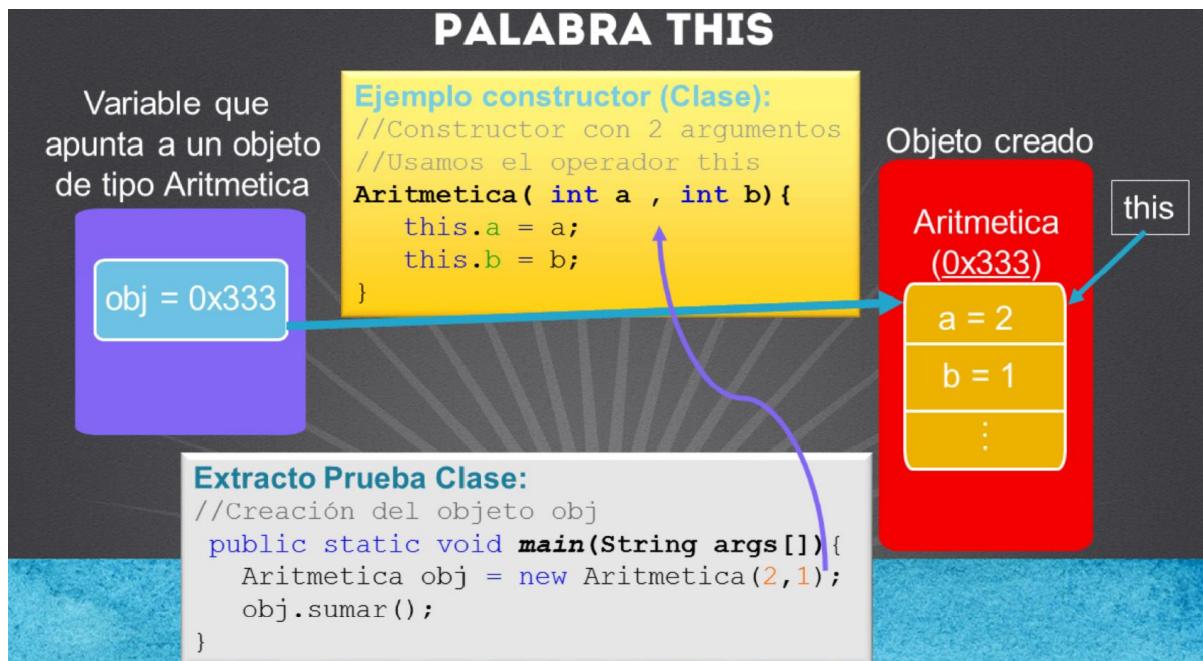


Los constructores, aunque son muy similares a los métodos, tienen ciertas diferencias y características, tales como:

- Estos métodos sólo se puede ejecutar al momento de la creación de un objeto, no es posible utilizarlos después.
- No devuelven ningún valor.
- El nombre del constructor es idéntico al nombre de la clase, así es como podemos identificar si una clase tiene constructores (siempre respetando las mayúsculas y minúsculas).
- Por defecto en Java, se crea un constructor sin argumentos, conocido como constructor vacío. Este constructor lo agrega en automático el compilador de Java a nuestra clase, sin embargo si nosotros definimos un constructor distinto al constructor vacío, es decir, con argumentos, entonces Java ya no agrega el constructor vacío y es nuestra responsabilidad agregar el constructor vacío a nuestra clase si fuera necesario.
- El constructor vacío es necesario para crear un objeto, recordemos la sintaxis básica general para crear un objeto.
 - `TipoClase objeto = new TipoClase();`
 - Si observamos al final de la línea de código estamos abriendo y cerrando paréntesis, ese es precisamente el constructor vacío que el compilador agregó por nosotros siempre y cuando no agreguemos constructores con otros argumentos. Por lo que ahora ya podemos entender que después de la palabra `new` realmente lo que estamos colocando es el nombre del constructor que deseamos llamar, pudiendo tener argumentos o no.

10.- ALCANCE DE VARIABLES

PALABRA THIS



En ocasiones un método necesita hacer referencia al objeto con el que estamos trabajando actualmente. Para esta tarea Java agregó la palabra reservada this. La palabra this es un operador el cual nos permite acceder al objeto actual (la clase con la cual estamos trabajando), y nos servirá, entre otras cosas, para acceder a los atributos o métodos de una clase. Con esto podemos hacer una diferencia entre los argumentos recibidos en un método y los atributos de una clase. Como podemos observar en la lámina, tenemos un código que tiene un atributo llamado a y b, y también el Constructor de la clase recibe dos argumentos llamados a y b. Para hacer diferencia entre estas dos variables (atributos de la clase y los argumentos recibidos en el método) podemos utilizar la palabra this como sigue:

```
//Constructor con 2 argumentos. Usamos el operador this
Aritmetica( int a , int b){
    this.a = a;
    this.b = b;
}
```

Como podemos observar, para hacer diferencia entre los valores recibidos en el Constructor y los atributos de la clase, podemos utilizar el operador this. Esto es solo un ejemplo del uso del operador this, pero básicamente nos permite acceder a los atributos y métodos del objeto actual con el cual estamos trabajando.

Dentro de los constructores o métodos de una clase, el operador this hará siempre referencia al objeto que fue invocado. Si observamos el código, tanto el argumento recibido como el atributo de la clase se llaman exactamente igual, por lo tanto toma prioridad el argumento sobre el atributo de la clase, a esto se le conoce como ocultamiento del atributo de la clase. Y para resolver este problema basta con utilizar el operador this antes de la variable, tal como si accedíramos al atributo de una clase por medio del operador punto.

ALCANCE DE UNA VARIABLE

```
package aritmetica;

public class Aritmetica {

    int a;
    int b;

    int sumar(int arg1, int arg2) {
        int resultado = arg1 + arg2;
        return resultado;
    }
}
```

Variables de clase

Variables de Clase:

- Pueden usarse en cualquier método de la clase
 - Se inicializan con valores por default

Variables Locales

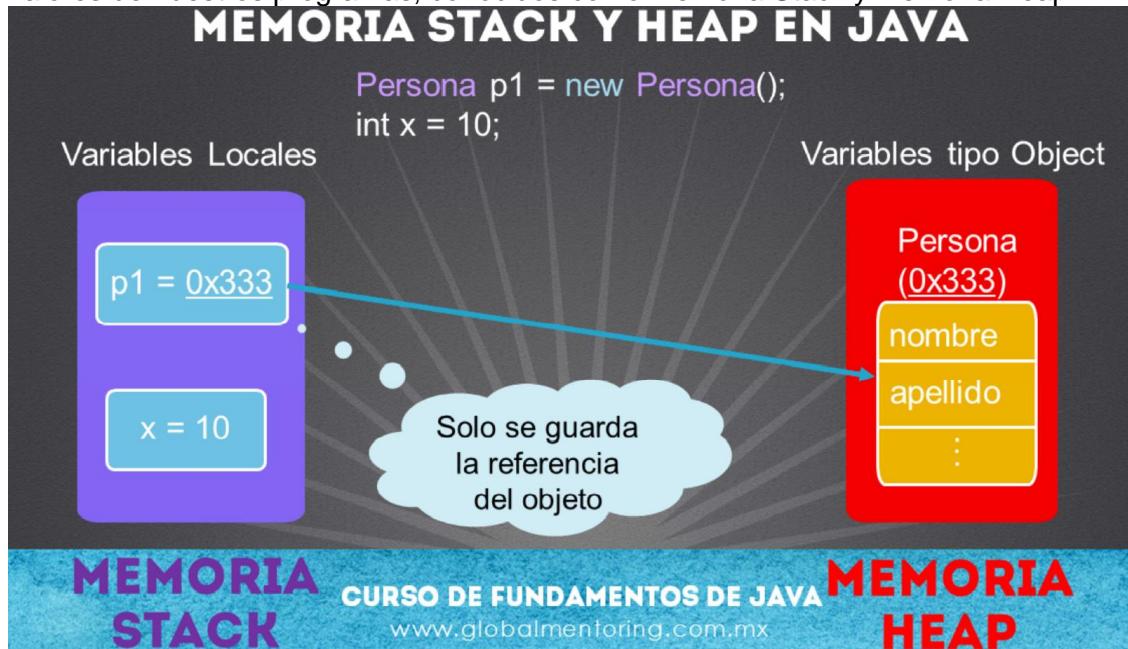
Variables Locales:

- Se pueden usar sólo en el método que se definen
 - Se deben inicializar

Las variables locales, ocultan a las variables de clase, y si queremos utilizar las variables de clase en un método que ha definido variables locales con el mismo nombre, entonces debemos utilizar el prefijo `this` para poder acceder a las variables de clase en lugar de los atributos locales.

11.- MEMORIA STACK Y HEAP EN JAVA

Como en muchos lenguajes de programación, la memoria RAM (Random Access Memory) se utiliza para almacenar la información de nuestro programa mientras éste se ejecuta. Java ejecuta un proceso, y a su vez internamente existen dos clasificaciones para almacenar los valores de nuestros programas, conocidos como memoria Stack y Memoria Heap.



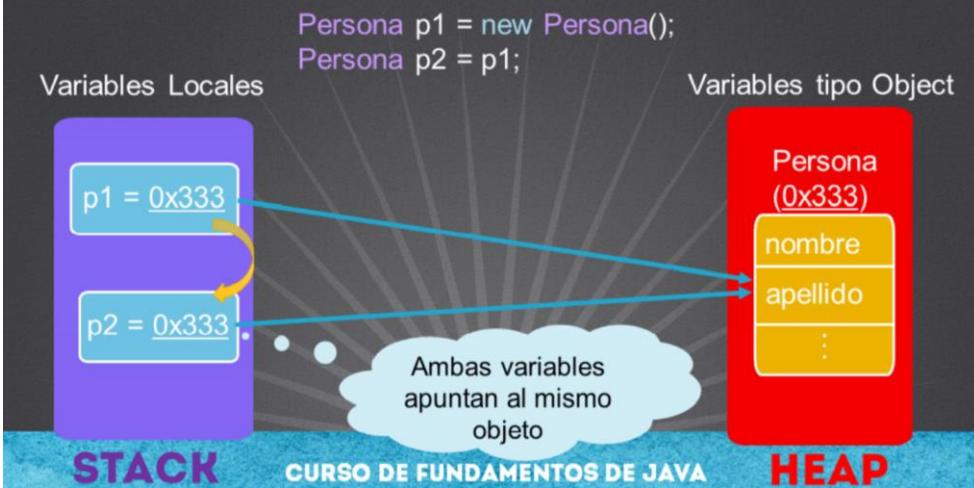
La memoria Stack se utiliza para almacenar las variables locales y las llamadas de funciones en Java. Las variables almacenadas en este espacio de memoria normalmente tienen un periodo de vida corto, únicamente mientras termina la función o método en el que se están ejecutando.

La memoria Heap se utiliza para almacenar los objetos Java, incluyendo sus atributos. Los objetos almacenados en este espacio de memoria normalmente tienen un tiempo de duración más prolongado.

En la figura podemos observar que la referencia del objeto se representa por un valor hexadecimal (0x333), el cual contiene la dirección de memoria donde está almacenado el objeto, y por lo tanto la variable local `p1` almacena únicamente esta referencia de memoria.

De esta manera el recolector de basura (garbage collector) puede buscar aquellos objetos en la memoria heap que ya no estén siendo referenciados por ninguna otra variable y finalmente liberar el espacio en memoria que ocupaba dicho objeto.

ASIGNANDO REFERENCIAS DE MEMORIA



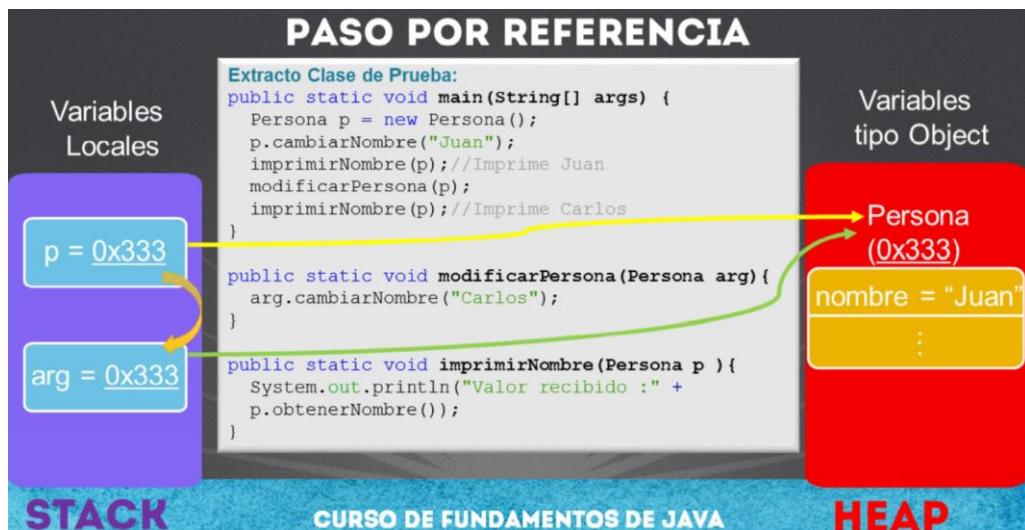
Es importante observar que no se está creando un segundo objeto, ya que solamente hay una llamada de la palabra new, y por lo tanto solo hay un objeto creado. Lo que almacena la variable p2 es sólo el valor de la referencia en memoria del objeto Persona creado. Cada que utilizamos la palabra new recibiremos una nueva referencia de memoria, es importante notar esto, ya que hasta ese momento es que se creará un nuevo objeto y no antes.

12.- PASO POR VALOR Y PASO POR REFERENCIA



Por ejemplo en el código mostrado podemos observar un ejemplo de paso por valor, también conocido como valores de tipo primitivo. Lo que observamos en el código es que al crear una variable, en este caso `x`, y asignarle un valor, por ejemplo 10, esta variable al ser modificada en otro método su valor original no se ve modificado, ya que solamente paso una copia de su valor original.

La manera de cambiar el valor de la variable `x` sería cambiándolo dentro del mismo método. Sin embargo en este ejemplo el método llamado `cambiarValor` intenta realizar un cambio a la variable `x`, pero como está fuera del alcance de este método acceder a esta variable, únicamente puede cambiar el valor de su variable local llamada `i`. Y al cambiar el valor de esta variable y regresar al método que hizo la llamada original



Al utilizar un objeto como parámetro y enviarlo a un argumento de un método lo que estamos haciendo es enviando la referencia del objeto que se está deseando utilizar, y en lugar de utilizar una copia del valor del objeto, lo que realmente estamos haciendo es

apuntar al mismo objeto, con la finalidad de modificarlo directamente, sin necesidad de hacer una copia.

En el código podemos observar la variable p de tipo Persona, esta clase la crearemos en la sección de ejercicios, pero básicamente tiene un atributo llamado nombre de tipo String, y lo que podemos observar es que se modifica el valor del atributo nombre, esto debido a que la variable local llamada arg apunta al mismo objeto creado en la memoria Heap de tipo Persona, y por lo tanto al modificar el atributo del mismo objeto, entonces se modifica el objeto creado originalmente, y podemos acceder a este cambio incluso desde otros métodos.

De esta manera podemos observar como en Java cuando trabajamos con objetos, realmente lo que estamos proporcionando es el valor de la referencia de memoria, en este caso el valor de 0x333, y con esta referencia accedemos directamente al objeto, y así podemos modificarlo. Finalmente estas modificaciones las podremos acceder incluso fuera del método donde fue creado originalmente el objeto, o desde el método donde se creó dicho objeto.

13.- USO DE LA PALABRA RETURN

1. Se encuentra la palabra return

Es utilizar la palabra return, y una vez que se ejecuta esta instrucción, se regresa el control al método que hizo la llamada previa. La palabra return puede no regresar ningún valor, es decir, solo colocar la palabra return, aunque esto no es necesario ya que Java si no se indica otra cosa ejecutará el final del método y regresará el control al método que hizo la llamada. La otra opción y es como normalmente se utiliza la palabra return, es agregando un valor, el cual corresponde con el tipo definido en la firma del método en cuestión.

Posteriormente veremos ejemplos del uso de la palabra return a más detalle.

2. Se llega al fin del método

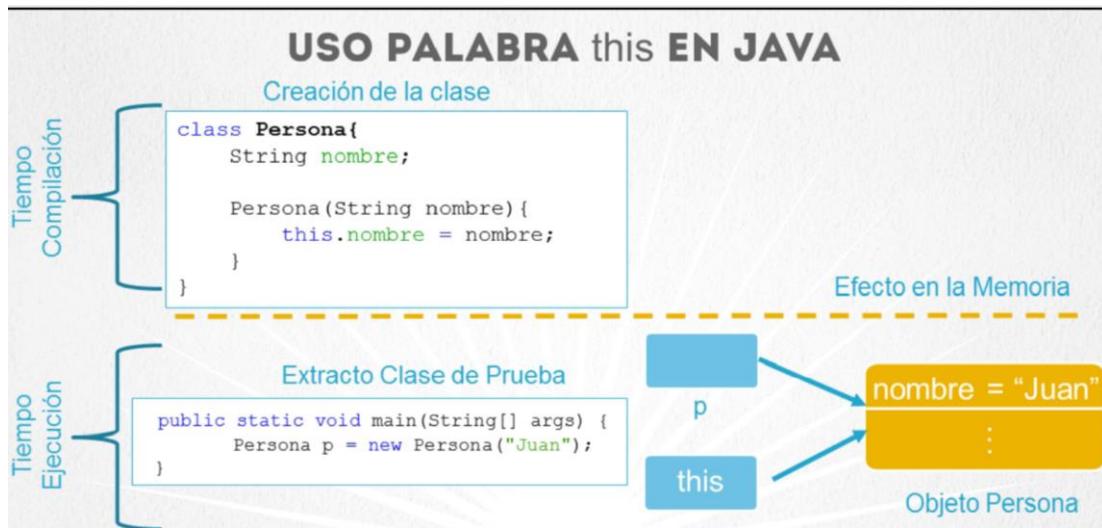
Es cuando llegamos al final de un método, no hay necesidad de colocar la palabra return de manera obligatoria, sino que una vez que se llega al final del método se hace return de manera implícita y con ello se obtiene el mismo resultado que si colocáramos la palabra return sin argumentos.

3. Ocurre un error

La forma de salir de un método es si ocurre un error, esto provoca que el método termine de manera anormal, y ya sea que se procese esta excepción o no, esto provocará que la ejecución del método concluya. El manejo de excepciones en Java es un tema que se tratará en otro curso a detalle.

14.- USO DE LA PALABRA this EN JAVA

- Es una referencia implícita que se está ejecutando.
- Es útil para evitar la ambigüedad entre las variables de clase y las locales
- Permite a un objeto enviarse él mismo como parámetro



Uso de los usos más comunes de la palabra this, como ya hemos comentado, es para evitar la ambigüedad entre los atributos de las clases (miembros de clase) y las variables locales o argumentos de un método. En el ejemplo podemos observar cómo estamos declarando una clase llamada Persona, la cual tiene un constructor que recibe un atributo llamado nombre, el cual es idéntico al nombre del atributo de la clase. Para hacer la diferencia entre ambos nombres, podemos utilizar la palabra this la cual hace referencia al atributo de la clase, de esta manera podemos nombrar de manera idéntica los argumentos de un método y diferenciar los atributos de la clase por medio del operador this.

En la figura podemos observar como tanto la variable p como el operador this apuntan al mismo objeto al momento de llamarse el constructor de la clase. Una vez que termina de ejecutarse el constructor de la clase el operador this apunta a otro objeto, y si de nuevo cuenta se vuelve a ejecutar algún método de la clase Persona, el operador this apuntará al objeto creado, es decir que el operador this siempre apuntará al objeto que se está ejecutando actualmente.

USO PALABRA this COMO REFERENCIA

Uso palabra this como referencia:

```
1 public class PalabraThis {
2     public static void main(String[] args) {
3         Persona p = new Persona("Juan");
4     }
5 }
6
7 class Persona {
8     String nombre; //atributo de la clase
9
10    Persona(String nombre) {
11        this.nombre = nombre; //this es el objeto Persona (actual)
12
13        //Imprimimos el objeto persona
14        Imprimir i = new Imprimir();
15        i.imprimir(this); //this es el objeto Persona (actual)
16    }
17 }
18
19 class Imprimir {
20
21     public void imprimir(Object o) {
22         System.out.println("Imprimir parametro: " + o); //el parametro es el objeto persona
23         System.out.println("Imprimir objeto actual (this): " + this); //this es el objeto imprimir (actual)
24     }
25 }
```

En la línea 3 creamos un objeto de tipo Persona. Hasta este punto el apuntador this no ha sido creado ni apunta a ningún objeto. Es sólo hasta que se ejecuta la línea 3 que sea crea el objeto de tipo persona y entonces se crea el apuntador this y apunta al objeto Persona recién creado.

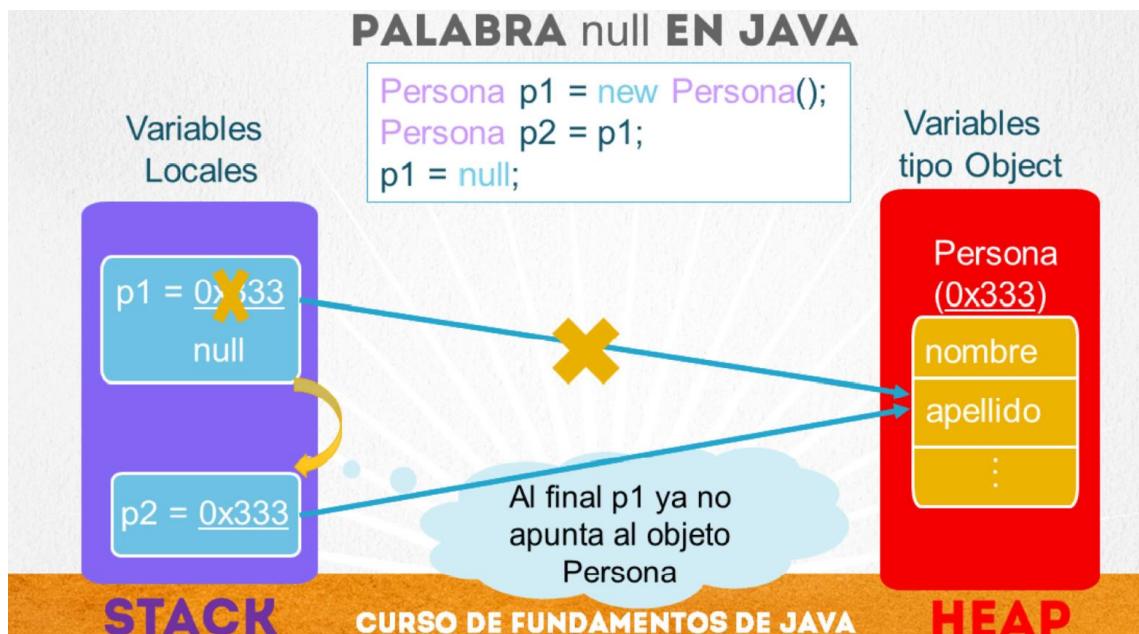
En el siguiente paso se ejecuta la línea 10, en la cual recibimos en la variable local de tipo nombre el valor de Juan, pero para poder diferenciar entre el argumento y el atributo de la clase utilizamos por primera vez la palabra this, de esta manera hacemos referencia al atributo de la clase.

Posteriormente creamos un nuevo objeto de la clase Imprimir, pero no es hasta que se ejecuta el método imprimir que el apuntador this cambiará de apuntar al objeto Persona al objeto Imprimir. Por ello en la línea 15 pasamos como parámetro el objeto Persona actual utilizando la palabra this, y cuando se ejecuta la línea 21 ocurren dos cosas importantes, por un lado el operador this deja de apuntar al objeto persona y el argumento que recibe el método imprimir es objeto Persona puesto que this en la línea 15 aún apuntaba al objeto persona.

Todas las clases en Java heredan de la clase Object, y aunque lo veremos en otro tema, la clase Object nos sirve como una clase comodín para poder recibir cualquier tipo Java como argumento, en este caso la variable Object creada almacena la referencia de la clase Persona creada anteriormente, y cuya referencia fue enviada en la línea 15 por medio del apuntador this.

Por ello en la línea 22 se imprime un objeto de tipo Persona, y en la línea 23 se imprime un objeto de tipo Imprimir, ya que recordemos que this ahora está apuntando al objeto que se está ejecutando actualmente, el cual es de tipo Imprimir.

15.- PALABRA null EN JAVA



La palabra `null` en Java la utilizamos con el objetivo de indicar que aún no se le ha asignado ninguna referencia de ningún objeto a una variable de tipo `Object`.

No es posible asignar el valor `null` a una variable de tipo primitivo

Como podemos observar en la figura, en primer lugar creamos la variable `p1` de tipo `Persona`, la cual se crea en la memoria heap y se le asigna la ubicación de memoria `0x333`.

Posteriormente creamos la variable `p2` y la asignamos la referencia almacenada por la variable `p1`, es decir, que ahora tanto la variable `p1` como la variable `p2` apuntan al mismo objeto y ambas variables pueden acceder a él.

Finalmente la variable `p1` decidimos que ya no vamos a utilizarla, y para ello asignamos el valor de `null`, esto significa que pierde la referencia del objeto creado y por lo tanto solamente la variable `p2` podrá ahora acceder al objeto `Persona` asignado en la ubicación de memoria `0x333`.

Esto significaría que el objeto `Persona` ya no lo apunta ninguna variable, y por lo tanto queda inaccesible. Desde el punto de vista del recolector de basura de Java sería un objeto candidato para ser eliminado de la memoria, ya que ninguna variable puede accederlo y por lo tanto es inservible, ya sólo queda que sea eliminado de la memoria Heap por medio del proceso de recolector de basura llamando al método `System.gc()`, es decir, garbage collector o recolector de basura.

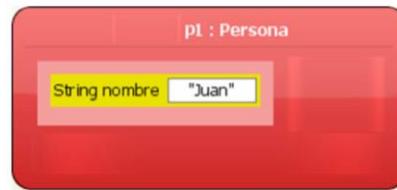
Sin embargo, si ya no vamos a utilizar un objeto, es conveniente y buena práctica que asignemos el valor de `null` a la variable que estaba apuntando al objeto creado previamente, con esto una vez que inicie el proceso de recolección de basura el objeto creado será candidato a ser eliminado.

Finalmente, podemos darnos cuenta que las variables creadas en la memoria Stack son temporales y de menor duración, y una vez concluido el método y/o programa en el que fueron creadas dichas variables, éstas se destruyen. Sin embargo las variables creadas en la memoria Heap, que son objetos Java, tienden a durar más tiempo en la memoria, y se destruyen hasta que se concluye el proceso de la máquina virtual de Java completo.

EJEMPLO PALABRA null EN JAVA

Uso palabra null:

```
1  public class PalabraNull {  
2  
3      public static void main(String[] args) {  
4          Persona p1 = new Persona("Juan");  
5          System.out.println("Nombre p1:" + p1.obtenerNombre());  
6  
7          Persona p2 = p1; //p2 apunta al mismo objeto que p1  
8          System.out.println("Nombre p2:" + p2.obtenerNombre());  
9  
10         //Hacemos que p1 ya no apunte al objeto p1  
11         p1 = null;  
12  
13         //Comprobamos que p2 sigue accediendo al objeto  
14         System.out.println("Nombre p2:" + p2.obtenerNombre());  
15     }  
16  
17     class Persona{  
18         String nombre; //valor por default es null  
19  
20         public Persona(String nombre){  
21             this.nombre = nombre;  
22         }  
23  
24         public String obtenerNombre(){  
25             return this.nombre; //Uso opcional de this  
26         }  
27     }  
28 }
```



Podemos observar el código de la lámina, en la cual estamos creando un objeto de tipo Persona, y la referencia la asignamos inicialmente a la variable p1.

En la línea 5 mandamos a imprimir el nombre del objeto Persona cuya referencia está almacenada en la variable p1. Y podemos observar que el valor es Juan.

Posteriormente en la línea 7 creamos la variable p2, esta variable le asignamos el valor de p1, es decir, ahora p2 también apunta al mismo objeto creado en la línea 4. Esto se comprueba en la línea 8 donde la variable p2 imprime el mismo nombre de la persona asignado en p1.

Posteriormente utilizamos la palabra null para indicar que la variable p1 NO apunta a ningún objeto, y de hecho si quisiéramos ejecutar el método de p1.obtenerNombre() nos arrojaría un error ya que esta variable ya no puede acceder ni a los métodos ni atributos del objeto Persona.

El uso de null y la razón por la cual nos marca este error conocido como NullPointerException, el cual será uno de los más comunes cuando trabajamos con Java, y básicamente significa que estamos tratando de acceder a un método o atributo de una clase en la cual la variable aún no se le ha asignado una referencia de un objeto válido y cuyo valor es null

Finalmente en la línea 14 comprobamos que la variable p2 continua accediendo sin problemas al objeto Persona y puede seguir imprimiendo el nombre asignado a este objeto.

16.- ENCAPSULAMIENTO EN JAVA

- El estado de un objeto esta generalmente oculto.

Esta característica nos permite aislar los datos de nuestros objetos del acceso de otros objetos externos, y de esta manera restringir el acceso directo a los atributos o métodos que no deseemos permitir, ya que el estado de un objeto está generalmente oculto. Podemos entender por estado de un objeto como los valores actuales de cada uno de los atributos del objeto, y cualquier cambio en estos valores cambia el estado interno del objeto.

- Esto se conoce como encapsulamiento
- Java utiliza modificadores de acceso para definir esta característica.
- Existe cuatro modificadores de acceso en Java, los cuales son: private, package o default, protected

Modificadores de acceso

- Firma de un método:
`modificador_acceso otros_modificadores nombreMétodo(listaArgumentos...);`
- Los modificadores de acceso que estudiaremos en ésta lección son: private y public
- private permite acceder sólo desde la misma clase al método o atributo marcado en este modificador
- public permite acceder desde cualquier clase a cual método o atributo definido con este modificador

EJEMPLO DE ENCAPSULAMIENTO

Ejemplo de encapsulamiento:

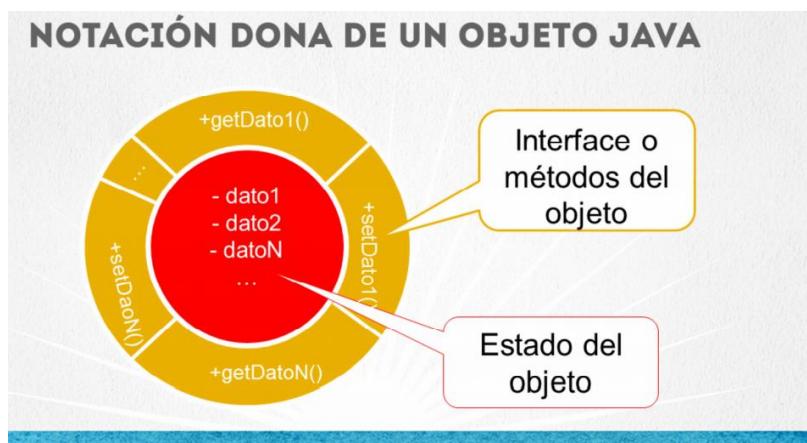
```
1 public class PruebaEncapsulamiento {  
2  
3     public static void main(String[] args) {  
4         //Creamos el objeto  
5         Persona p1 = new Persona();  
6         //Modificamos el atributo nombre  
7         p1.setNombre("Juan");  
8         //Accedemos al atributo nombre  
9         System.out.println("Nombre:" + p1.getNombre());  
10    }  
11}  
12  
13 class Persona {  
14     //Atributo privado  
15     private String nombre;  
16     //Método público para acceder al atributo nombre  
17     public String getNombre(){  
18         return nombre;  
19     }  
20     //Método público para modificar al atributo nombre  
21     public void setNombre(String nombre){  
22         this.nombre = nombre;  
23     }  
24 }
```

En la lámina podemos observar un código más formal para el concepto de encapsulamiento, es decir, hemos convertido nuestros atributos a privados (línea

15), y por cada atributo hemos agregado dos métodos, uno para acceder al atributo (getNombre - línea 17), y otro para modificar el atributo (setNombre - línea 21).

Los métodos de tipo accessors permiten recuperar el valor del atributo de una clase y se utiliza el prefijo is en caso de ser de tipo boolean y get para los demás tipos de datos, posteriormente lleva el nombre del atributo de igual manera respetando la notación de mayúsculas/minúsculas en Java.

En este ejemplo debemos entender que la clase de prueba es la clase pública llamada PruebaEncapsulamiento, y que la clase en la que se aplicó el concepto de encapsulamiento fue sobre la clase Persona. Es decir que son clases distintas, y por ello el código que tenemos en el método main (líneas 4 a 9) es código que está fuera de la clase Persona y allí radica el concepto de encapsulamiento, es decir, evitar que clases externas a la clase persona puedan acceder a los atributos de la clase Persona directamente, sin importar que sea una clase declarada en un mismo archivo, son clases distintas.



Para evitar el acceso directo de otras clases a los datos de la clase que deseamos encapsular lo primero que debemos hacer, según hemos comentado, es agregar el modificador de acceso private a nuestros atributos de clase, de esta manera únicamente los métodos de la misma clase son los que podrán acceder (leer/get) y manipular (modificar/set) los datos de la clase que estamos creando.

Por otro lado, ¿Cómo podrán acceder una clase externa a los datos de nuestros objetos encapsulados? La respuesta es creando **métodos públicos por cada atributo privado**, los cuales puedan tanto modificar como recuperar el valor de nuestros datos o atributos de clase. A esto se le conoce como la **interface** del objeto, debido a que es a través de estos métodos que nos estaremos comunicando con el objeto creado y así poder leer el estado de cada dato y/o modificar también sus datos. Normalmente se crean dos métodos públicos por cada atributo privado, un método set y otro get.

Los métodos que modifican los datos se conocen como **mutators** y llevan el prefijo set (colocar) seguido del nombre de la variable en notación de

mayúsculas/minúsculas. Por otro lado, los métodos que leen la información de los datos se conocen como **accessors** y llevan el prefijo de **get** (obtener) seguido del nombre de la variable, excepto en los casos en que la variable sea de tipo boolean, entonces el prefijo en lugar de get será **is** (es).

Ahora, ¿Por qué evitar la modificación directa del estado de un objeto en Java, y en general en la programación orientada a objetos? Lo que buscamos es que tengamos un control sobre la información y estado de nuestros objetos, de tal manera que si queremos por ejemplo modificar el dato1, podamos aplicar una validación sobre la información que queremos colocar en el dato1, y este sea un valor adecuado, como puede ser la restricción de recibir sólo números, entonces, antes de hacer la actualización del valor del dato1 podamos verificar que efectivamente el valor que se va a colocar sea un valor numérico, esto sólo por poner un ejemplo sencillo, pero la idea es que otros objetos no puedan directamente modificar el estado de nuestros objetos si no lo deseamos, y de esta manera el concepto de encapsulamiento en Java nos permite lograr esto, ocultando cualquier validación pero aplicándola en todo momento, respetando en todo momento las reglas que apliquen al estado del objeto.

En la lámina podemos observar que los atributos tienen un símbolo de – (menos), y que los métodos tienen un símbolo de + (mas), esto es parte de la notación de este diagrama de dona o de objetos, y lo que nos indica es que el signo de – es el modificador de acceso private, y el signo de + es el modificador de acceso public, de esta manera podemos observar gráficamente los elementos privados y públicos en nuestro objeto Java.

17.- CONTEXTO ESTÁTICO



Para comenzar a crear objetos de una clase, primero se debe cargar la clase en memoria por medio de lo que se conoce como ClassLoader (cargador de clases). Aquí entra por primera vez el concepto de contexto estático, y podrá ser utilizado hasta que la clase se elimine de la memoria, que normalmente ocurre cuando se detiene el proceso de la máquina virtual de Java, es decir, que la clase está cargada durante todo el proceso de la ejecución de nuestra aplicación.

Una vez que la clase ya está cargada en memoria, es posible empezar a crear objetos de dicha clase. A esto se le conoce como el contexto dinámico, ya que en este momento ya es posible crear objetos y empezar a interactuar entre los mismos.

En resumen, el contexto estático se carga primero y por lo tanto las clases que se vayan a utilizar, y posteriormente se crea el contexto dinámico y por lo tanto ya se pueden crear los objetos de las clases que se hayan cargado en memoria. Como podemos observar en la lámina, el contexto estático tiene una mayor duración que el contexto dinámico, y de hecho el contexto estático incluye el contexto dinámico, pero no al revés.

El operador `this` sólo tiene uso cuando se ha creado un objeto de una clase, por lo tanto no es posible utilizarlo en el contexto estatico.

PALABRA STATIC



Podemos utilizar la palabra **static** para interactuar con el contexto estático. Por ejemplo, si definimos un atributo o un método como estático, lo que estamos indicamos es que el atributo o método pertenecen a la clase y no al objeto.

Por ejemplo, si creamos un atributo sin usar la palabra static, que es como normalmente los definimos, cada que creamos un objeto se creará también una variable asociada al objeto que se crea, pero si definimos el atributo como estático, estamos indicando que el atributo solo se crea una vez, sin importar cuantos objetos se creen, sólo habrá una variable la cual se asocia a la clase y no al objeto. Y si un objeto accede al valor de la variable estática leerá el mismo valor que los demás objetos, y si un objeto modifica el valor estático, todos los demás objetos accederán al mismo valor ya que este valor está almacenado en la clase y no en los objetos de dicha clase.

Con el uso de la palabra static ya sea en el atributo o en el método que deseamos acceder podremos usar directamente el atributo o método sin necesidad de generar un objeto de la clase, solo colocando el nombre de la clase, el operador punto, y finalmente el nombre del atributo o método estático definido en la clase. Normalmente los atributos o métodos estáticos si deseamos que sean accedidos desde otras clases deberán contener el modificador de acceso public para que no tengan ningún problema en ser accedidos.

STATIC Y THIS

Uso de this y el contexto estático:

```
public class EjemploThisStatic {  
  
    public static void main(String[] args) {  
        //this no puede usarse dentro de un contexto estático  
        //ya que solo tiene sentido cuando se ha creado un objeto  
        this.imprimir("Hola"); //Marca error  
  
    }  
  
    //Metodo NO estatico  
    public void imprimir(String s){  
        System.out.println("Valor recibido: " + s);  
    }  
}
```

El operador this se asocia con el contexto dinámico, y debido a que el contexto estático carga primero que el contexto dinámico, el operador this no funciona en el contexto estático.

Es por esta razón que en el método main o cualquier otro método estático, no es posible utilizar el operador this.

Ahora, podemos acceder a un método dinámico desde un método estático siempre y cuando creemos un nuevo objeto.

EJEMPLO CODIGO ESTATICO

```
public class EjemploStatic {  
    public static void main(String[] args) {  
        Persona p1 = new Persona("Juan");  
        System.out.println("Personal: " + p1);  
  
        //Imprimimos el contadorPersonas  
        System.out.println("No. Personas: " + Persona.getContadorPersonas());  
    }  
}  
  
class Persona {  
  
    private String nombre;  
    private int idPersona;  
    private static int contadorPersonas;  
  
    public Persona(String nombre){  
        //Cada que creamos un objeto persona incrementamos el contador para obtener un nuevo idPersona  
        contadorPersonas++;  
        //Asignamos el nuevo valor al idPersona  
        idPersona = contadorPersonas;  
        //Asignamos el nombre recibido  
        this.nombre = nombre;  
    }  
  
    public static int getContadorPersonas(){  
        return contadorPersonas;  
    }  
}
```

Por un lado declaramos una variable de tipo static llamada contadorPersonas. Al declarar esta variable como estática, lo que buscamos es que cada vez que creamos un objeto de tipo Persona, vamos a incrementar el valor de esta variable. Esto lo podemos lograr utilizando el constructor de la clase, ya que cada que creamos un objeto de tipo Persona, se mandará llamar el constructor de la clase, y allí podemos incrementar esta variable con cada llamada realizada al constructor (línea 20).

Una vez que se ha incrementado esta variable estática, este valor aplica para todos los objetos, por lo tanto cada que incrementamos esta variable puede ser consultada por todos los objetos que creemos y de esta manera es que esta variable no comienza desde cero

cada que creamos un nuevo objeto, sino que se queda en el último valor y se incrementa una vez más, esto debido a que esta variable pertenece a la clase y no a los objetos.

Ahora, en el método main, el cual es un método estático, vamos a hacer uso de la clase Persona. En primer lugar crearemos un objeto de tipo Persona, y asignamos el valor de Juan al atributo nombre con ayuda del constructor de la clase.

Desde el momento que estamos utilizando la palabra new, quiere decir que ya estamos trabajando con el contexto dinámico, esto permite trabajar ya con el contexto dinámico (objetos) y el contexto estático (clases).

En este código estamos omitiendo la implementación del método `toString` para la clase Persona, el cual se debe agregar para observar el estado del objeto persona impreso en la línea 5.

Finalmente, una vez que hemos terminado de crear objetos de tipo persona, podemos mandar a imprimir el número total de objetos Persona que se han creado, y aquí es donde podemos observar la notación que hemos platicado, para mandar a llamar un método estático público de una clase basta con colocar el nombre de la clase, el operador punto y el nombre del método estático, es decir que no debemos utilizar el objeto, de hecho no es buena práctica y si insistimos en usar una variable de tipo object para mandar a llamar un método estático nos mostrará un warning, avisándonos

18.- HERENCIA EN JAVA

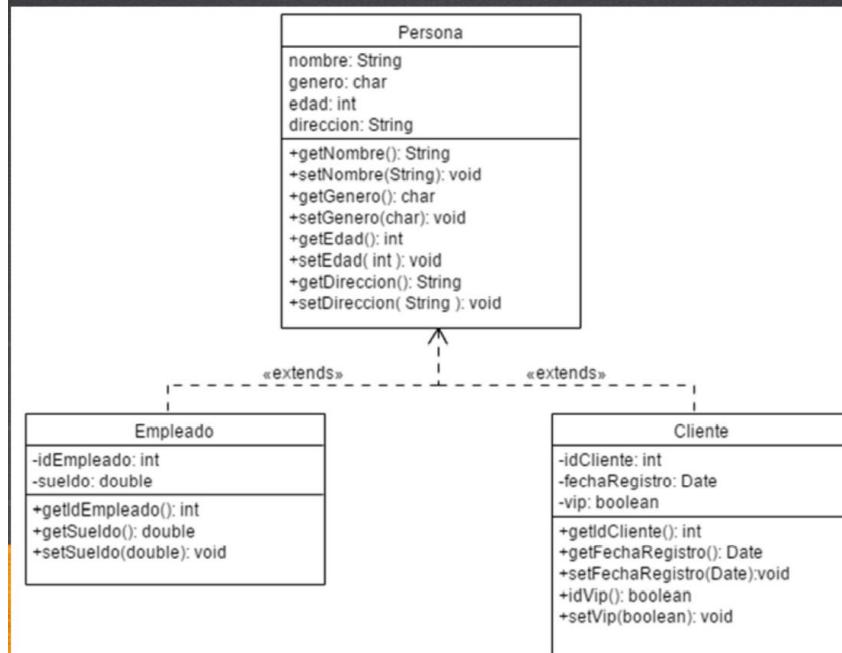
Una de las formas en que entendemos la herencia, son las características que se mantienen de generación en generación en una familia. Por ejemplo, los abuelos tenían cabello rubio, y uno de los hijos hereda esta característica, a su vez el hijo tiene a su vez un hijo que también hereda el cabello rubio.

En la programación orientada a objetos, el concepto de herencia es exactamente igual. Definiremos una jerarquía de clases que nos permitirán heredar características entre clases Padre y clases Hijas.

La herencia nos permitirá representar características o comportamiento en común entre clases, permitiendo definir en la clase Padre los atributos o métodos que sean comunes a las clases hijas, las cuales heredarán estos atributos o métodos definidos en la clase Padre.

Lo anterior permite evitar duplicar el código entre la clase Padre y las clases Hijas, por lo que cumplimos con la reutilización de código que es uno de los principales objetivos de la POO.

EJEMPLO DE HERENCIA EN JAVA



Vamos a ver un ejemplo de herencia en Java. Podemos observar una clase llamada Persona, la cual como cualquier persona tiene muchas características, pero nos vamos a enfocar a unas cuantas. Una persona tiene un nombre, un genero, una edad y una dirección, vamos a capturar estas características en cada uno de los atributos que

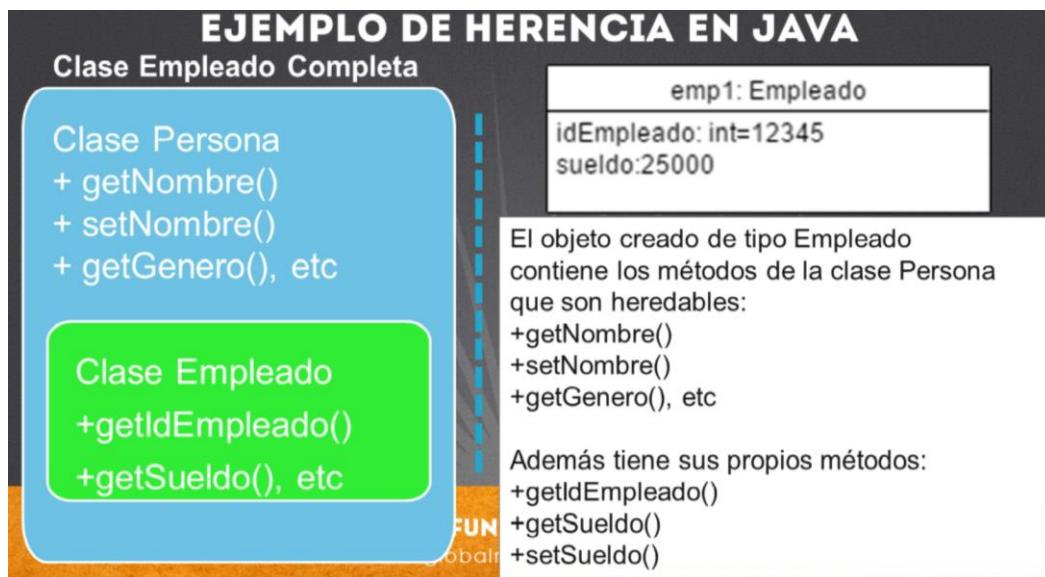
corresponden en la clase Java, puede tener muchos más, pero nos vamos a enfocar en estos para simplificar el ejemplo.

Por otro lado, si estamos tratando de modelar un sistema de ventas, podríamos tener algunas variantes de una persona en nuestro sistema, por ejemplo la clase de Empleado y Cliente.

En este ejemplo no estamos indicando cómo llegamos al diseño de estas clases, sin embargo el proceso general es identificar los actores u objetos (entidades que generalmente serán nuestras clases) que interactuarán en nuestro sistema, y a partir de las necesidades de nuestro sistema podemos ir anotando cuales son las características (atributos) y funcionalidad (métodos) que debe tener cada entidad registrada.

Una vez que se han encontrado estas entidades con sus atributos y métodos, vemos si existen características o funcionalidad en común, y allí es precisamente como llegamos a este tipo de diseños. Podemos observar que la clase Persona es la clase Padre y que las clases Empleado y Cliente extienden de la clase Persona, es decir, que heredan las características de la clase Persona.

Esto permite que no tengamos que repetir el código ya definido en la clase Persona, y una vez que creamos un objeto de tipo Empleado o Cliente, en automático también tendrá las características de la clase Persona.



Lo que visualizamos es que la clase Persona tiene atributos privados y varios métodos públicos. Los atributos privados no se heredan, al igual que los métodos privados. Es decir, mientras usemos el modificador de acceso private no se heredara el atributo o método marcado con este modificador.

En resumen, cuando visualicemos una clase que hereda de otra, podemos imaginarnos a esta clase como el conjunto de sus propios atributos y métodos, así como la suma de los atributos o métodos heredables de la clase o clases Padre, ya que puede recibir herencia

de una clase Padre, o una clase “Abuelo”, es decir, una clase Padre de su propia clase Padre.

SINTAXIS HERENCIA EN JAVA

Sintaxis Herencia en Java:

```
class Persona {...} //Definición de la clase Padre o Super Clase
class Empleado extends Persona {...} //Definición de la clase Hija o Subclase
class Gerente extends Empleado {...} //Definición de otra clase Hija o Subclase
```

Nota: TODAS las clases que no especifican de manera explícita un extends, entonces heredan de la clase Object, la cual es la clase core de Java.

Para definir la herencia en Java basta con utilizar la palabra extends en la definición de la clase Java e indicar el nombre de la clase Padre.

Ej. class Empleado extends Persona

Cabe aclarar que en Java la herencia es simple, esto quiere decir que sólo podemos heredar de una clase a la vez. Puede haber una jerarquía de clases hacia arriba, por ejemplo Clase Abuelo, Padre e Hija, pero una clase no puede heredar por decir de una clase Padre y una clase Madre, es decir, de dos clases al mismo tiempo, únicamente por jerarquía de clases.

Esto NO está permitido (NO se puede heredar de dos clases a la vez, en Java la herencia es simple):

class Gerente extends Empleado, Persona

Es importante indicar que TODAS las clases en Java heredan de la clase Object, de manera indirecta o directa. En el caso de la clase Persona, como no está indicando que hereda de alguna clase, es decir, no usa extends en la definición de su clase, se entiende que en automático hereda de la clase Object. Pero las clases Empleado o Gerente, debido a que extienden de otra clases, heredarán los métodos de la clase Object de manera indirecta, pero todas las clases en Java de una manera u otra heredan las características de la clase Object. Finalmente, la clase Object es la clase core o base de Java, y pertenece al paquete java.lang. Posteriormente hablaremos del tema de paquetes en Java.

USO DE SUPER Y THIS

```
class Persona {//Definición de la clase Padre o Super Clase

    //Constructor 1 argumento
    public Persona(String nombre){
        this.nombre = nombre;
    }

    class Empleado extends Persona {//Definición de la clase Hija o Subclase

        public Empleado(String nombre, double sueldo) {
            super(nombre); //Super debe ser la primera linea
            this.sueldo = sueldo;
        }
        //Prueba creación del objeto Empleado
        public static void main(String[] args) {
            Empleado e1 = new Empleado("Juan", 25000);
        }
    }
}
```

Si aplicamos el concepto de herencia la clase Hija o Subclase puede acceder a los atributos, métodos o constructores permitidos de la clase padre solo con utilizar la palabra super.

De esta manera podemos aprovechar código heredado y aprovecharlo por ejemplo para iniciar un objeto de la clase hija. Esto es muy común utilizarlo dentro del constructor para llamar desde el constructor de la clase hija al constructor de la clase Padre.

De hecho la palabra this se puede utilizar de manera idéntica pero para llamar a un constructor de la misma clase, y la palabra super para llamar a los constructores de la clase Padre. Ojo, super no solo permite llamar constructores de la clase padre, sino cualquier atributo, método o constructor heredable de la clase Padre.

En el ejemplo podemos observar que utilizamos la palabra super para mandar a llamar el constructor de la clase Padre desde la clase Hija.

toString Y SUPER

```
//Extractos de las clase Persona y Empleado
public class Persona {
    @Override
    public String toString() {
        return "Persona{" + "idPersona=" + idPersona
            + ", nombre=" + nombre
            + ", edad=" + edad + '}';
    }
}

public class Empleado extends Persona {
    @Override
    public String toString() {
        //Primero mandamos a llamar el método toString de Persona
        //para que podamos observar los valores de la clase Padre,
        //y despues imprimimos los valores de la clase hija
        return super.toString() + " Empleado{sueldo=" + sueldo + "}";
    }
}
```

En la lámina podemos observar un ejemplo para el uso del método `toString()`. El método `toString()` según hemos comentado, lo vamos a utilizar para mostrar el estado de un objeto, es decir, los valores de los atributos en cierto momento del tiempo de vida del objeto.

El método `toString` es un método heredado de la clase `Object`. La clase `Object` es la clase Padre de todas las clases Java, ya sea de manera explícita o implícita, según hemos comentado. Por ejemplo, en el caso de la clase `Empleado`, la clase `Object` sería la clase abuela de la clase `Empleado`, debido a que se ha definido que de manera explícita la clase `Persona` como su clase Padre, y debido a que la clase Padre no ha indicado que extiende de alguna clase, entonces se entiende que su clase padre es la clase `Object`. De esta manera la clase `Empleado` también recibe los métodos y atributos que llegase a heredar de la clase `Object`.

Para indicar que un método pertenece a una clase padre o de nivel superiores, como la clase `Object`, entonces debemos agregar la anotación `@Override`, esto quiere decir que estamos sobreescritiendo el comportamiento de un método de la clase padre o clases superiores.

Ahora, este método `toString()` nos sirve según hemos comentado, para convertir el estado de un objeto a una cadena. Pero para el caso que estamos mostrando, la clase `Persona` tiene su método `toString()` y lo ideal es que reutilizaremos este código para completar el método `toString` de la clase `Empleado`, ya que si combinamos el método `toString` de la clase `Persona` con el de la clase `Empleado`, entonces podremos mostrar el estado de todos los atributos de la clase `Empleado` y de su clase padre `Persona`.

Cómo hacemos esto? Como observamos en el código, lo que hacemos es utilizar la palabra `super`, y debido a que `toString` es un método público redefinido (sobreescrito) en la clase `Persona`, entonces podemos acceder a este método de la clase padre por medio de la palabra `super`, el operador punto y el nombre del método que deseamos acceder del padre. Esto nos va a regresar una cadena con el estado del objeto `Persona`, y con ello solo basta concatenar el valor que aún no hemos colocado de la clase `Empleado`, por ello concatenamos el valor de la variable `sUELDO` y ahora si podemos regresar la cadena completa para que ahora el método `toString` de la clase `Empleado` pueda mostrar tanto su propio estado como el estado.

19.- SOBRECARGA DE CONSTRUCTORES

```
//Sobrecarga de Constructores en Java
public class Persona { // clase
//Constructores sin argumentos
Public Persona () {
}

//Constructor sobrecargado
public Persona ( String nombre int edad) {
this.nombre = nombre; // inicializador de variables clases
this.edad = edad; //inicializador de variables clases
}
```

Podemos decir entonces, que la sobrecarga de un Constructor es ofrecer más opciones para poder construir un objeto de una clase. En el código mostrado en la lámina podemos observar que en primer lugar se ha creado el constructor vacío Persona (línea 4), es decir, sin argumentos. Sin embargo muchas veces queremos ofrecer más opciones de construir un objeto de tipo Persona y que desde el momento de la creación obligue a proporcionar los datos de nombre y edad.

Por lo tanto, agregamos un constructor sobrecargado, con 2 argumentos (línea 9), por lo que tenemos Persona(String, int). Sin embargo, si agregamos un Constructor llamado Persona con los argumentos invertidos, es decir, Persona(int, String) entonces éste se considera otra sobrecarga de los constructores ya definidos. Es decir que el orden y el tipo de los argumentos sí importa.

En resumen, la sobrecarga de Constructores, es definir un Constructor con el mismo nombre de la clase, pero con distintos argumentos, considerando el tipo y orden de los argumentos. Esto se hace con el objetivo de brindar varias opciones para la creación de nuestros objetos según la clase que estemos codificando.

USO DE super SOBRECARGA DE CONTRUCYORES

```
//Uso de super en la sobrecarga de Constructores en Java
public class Empleado extends Persona { //clases
    private double sueldo; // variable clase

    public Empleado ( String nombre int edad double sueldo) {
        //Contructor
        //Susper debe ser la primera linea
        super(nombre, edad);
        this.suemdo =sueldo;
    }
}
```

Partiendo de la clase Persona que tenemos en la lámina anterior, definiremos la clase Empleado, la cual extiende de la clase Persona. Por lo tanto hereda todas las características que son heredables, es decir, que no son de tipo private.

Esto quiere decir que la clase Empleado tiene acceso al constructor público Persona de dos argumentos (String e int), y por lo tanto podemos apoyarnos de el para inicializar los atributos de la clase Persona. De hecho, por la forma en que está construida la clase Persona, es la única forma de inicializar los atributos de dicha clase.

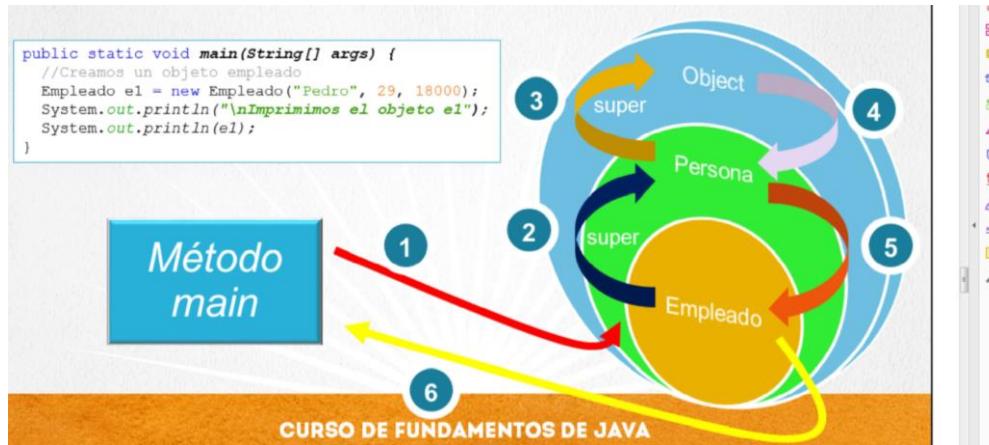
Por ejemplo, el atributo idPersona de la clase Persona, no existe forma de accederlo directamente, ni siquiera a través de algún método o un constructor. La única manera de accederlo e inicializarlo es mandando a llamar el constructor vacío, sin embargo, este constructor vacío de la clase Persona es privado, por lo tanto, no podemos usarlo desde la clase Empleado.

La única forma que tenemos de inicializar el objeto Persona es a través de la llamada al constructor público Persona(String nombre, int edad). Ahora, ¿la pregunta es cómo podemos acceder a este constructor?

La respuesta es utilizando la palabra super. Esta palabra nos permite acceder a los constructores, métodos o atributos de la clase padre siempre y cuando sean accesibles para nuestra clase. Cuando veamos a detalle el tema de modificadores de acceso veremos que existen otros modificadores además de public que nos permitirán acceder a Constructores, métodos o atributos de la clase padre.

Sin embargo para el ejemplo que estamos mostrando, la palabra super la estamos usando para inicializar los atributos de la clase Persona, de la cual extiende Empleado, si no lo hiciéramos de esta forma, estaríamos dejando sin asignar los valores respectivos a los atributos de la clase Persona, ya que todos sus atributos son privados, y se podrían asignar valores si existieran métodos de tipo mutator (set) para los atributos respectivos, pero si no existe este método mutator entonces se quedarían sin inicializar.

ORDEN DE LA LLAMADA DE CONSTRUCTORES

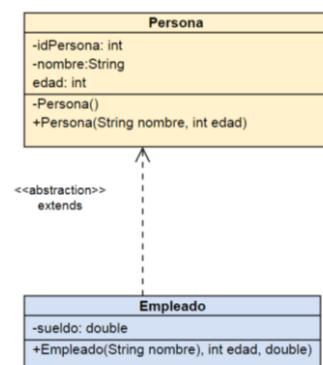


En primer lugar se manda a llamar el constructor del objeto que estamos creando. Al comenzar a ejecutar este constructor, se hace una llamada implícita a `super()`, es decir, a la clase Padre.

Todas las clases en Java heredan de la clase `Object` según hemos comentado, por lo tanto todos los constructores en un momento u otro hacen una llamada a `super()`, y eventualmente termina llamando al constructor vacío de la clase `Object`. Este constructor es el encargado de reservar memoria entre varias tareas más, sin embargo para esta lección basta con saber que es el constructor que se manda a llamar en todos los casos que se crea un objeto.

En pocas palabras los constructores hijos son los que inician la llamada, pero estos a su vez en su primera línea de código, de manera directa o indirecta llaman al constructor de la clase padre. Si no es especifica nada, entonces la llamada es al constructor vacío de la clase padre, es decir, `super()`;

DIAGRAMA UML



```

//Código completo de la clase Persona

public class Persona {
    private int idPersona ;
    private String nombre;
    private int edad;
    private static int contadorPersonas;

    private Persona (){
        this.idPersona = ++contadorPersonas;
    }
//Sobre carga de Constructor
    public Persona(String nombre, int edad){
        this();
        this.nombre = nombre;
        this.edad = edad;
    }

    public int getIdPersona() {
        return idPersona;
    }

    public void setIdPersona(int idPersona) {
        this.idPersona = idPersona;
    }

    public String getNombre() {
        return nombre;
    }

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

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }

    @Override
    public String toString() {
        return "Persona{" +
            "idPersona=" + idPersona +
            ", nombre=" + nombre + '\'' +
            ", edad=" + edad +
            '}';
    }
}

}

//Codigo de la clase de Empleado

public class Empleado extends Persona {
    private double sueldo;

    public Empleado (String nombre, int edad , double sueldo){
        super(nombre,edad);
        this.sueldo = sueldo;
    }

    public double getSueldo() {
        return sueldo;
    }

    public void setSueldo(double sueldo) {
        this.sueldo = sueldo;
    }

    @Override
    public String toString() {
        return super.toString() + " Empleado{" +
            "sueldo=" + sueldo +
            '}';
    }
}

}

//Codigo de la clase de SobreCargaConstructor

public class SobreCargaConstructores {
    public static void main(String[] args) {
        Persona persona1 = new Persona("Juan", 23);
        System.out.println("persona1 = " + persona1);

        Empleado empleado1 = new Empleado("Luis",35, 18888);
        System.out.println("empleado1 = " + empleado1);
    }
}

```

20.- SOBRE CARGA DE MÉTODOS

Sobrecarga de métodos en Java:

```
//Definimos primeramente el método suma  
int sumar(int a, int b){  
    return a + b;  
}  
  
//Si agregamos un método con el mismo nombre previamente definido  
//Pero con distintos argumentos entonces se obtiene la sobrecarga  
double sumar(double a, double b){  
    return a + b;  
}
```

Posibles variantes de la sobrecarga

● sumar(double a, double b) double
● sumar(double a, int b) double
● sumar(int a, double b) double
● sumar(int a, int b) int

Una de las reglas de la sobrecarga de métodos es que lo que observa el compilador para que un método cumpla con una sobrecarga válida es que los tipos de los argumentos sea distinto a los del método ya definido, esto incluye el orden de los argumentos, pero en ningún caso el compilador revisa que los nombres de los argumentos sea igual o no, de igual manera no revisa si el tipo de retorno es igual o no, es decir, que el tipo de retorno tampoco importa al momento de agregar un método que cumpla con la sobrecarga.

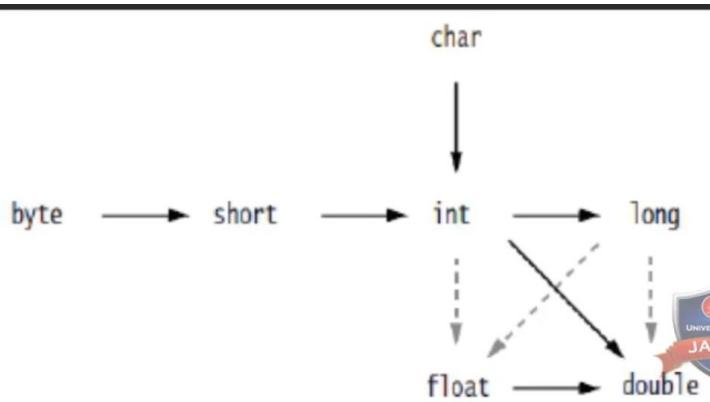
Entonces, para que una sobrecarga sea válida debe cumplir con lo siguiente:

- 1) El nombre del método debe ser igual al método que se desea sobrecargar.
- 2) Los argumentos del método deben ser distintos al método que se desea sobrecargar, únicamente se revisa el tipo y el orden en que se agregan, no se revisa el nombre del argumento.
- 3) El tipo de retorno no afecta si es igual o distinto al del método a sobrecargar.
- 4) El método a sobrecargar puede estar definido en nuestra clase o en alguna clase superior.
- 5) Los argumentos pueden ser de tipo primitivo o de tipo Object

En caso de aplicar alguna conversión de los tipos a utilizar, se aplicará la conversión automática superior que aplique, por ejemplo si hacemos uso de dos variables de tipo long, y hacemos una llamada al método sumar, no se llamará al método con argumentos de tipo int, sino al método sumar con argumentos de tipo double, ya que el tipo long se

convertirá al tipo superior de manera automática, es decir al tipo double, y quedará descartado el tipo int por ser un tipo de menores bits y menos capacidad para almacenar un dato de tipo long.

IMAGEN DE SOPORTE DE LAS VARIABLES PRIMITIVOS



// CODIGO DE LA CLASE OPERACIONES

```
public class Operaciones {  
  
    public static int sumar(int a, int b) {  
        System.out.println("metodo sumar(int, int)");  
        return a + b;  
    }  
  
    public static double sumar(double a, double b) {  
        System.out.println("metodo sumar(double, double)");  
        return a + b;  
    }  
  
    public static double sumar(int a, double b) {  
        System.out.println("metodo sumar(int, double)");  
        return a + b;  
    }  
  
    public static double sumar(double a, int b) {  
        System.out.println("metodo sumar(double, int)");  
        return a + b;  
    }  
}
```

// CODIGO DE LA CLASE SOBRECARGAMETODOS

```
public class SobreCargaMetodos {  
    public static void main(String[] args) {  
        System.out.println("Resultado 1: " +Operaciones.sumar(1,2));  
        System.out.println("Resultado 2: " +Operaciones.sumar(1.5,3));  
        System.out.println("Resultado 3: " +Operaciones.sumar(1,3.1568));  
        System.out.println("Resultado 4: " +Operaciones.sumar(4,4l));  
        System.out.println("Resultado 5: "+Operaciones.sumar(4.5d, 3.1));  
        System.out.println("Resultado 6: " +Operaciones.sumar(1.5,'a'));  
  
    }  
}
```

21.- PAQUETES JAVA

En Java lo que organizamos son clases, y básicamente nos permite agrupar clases ya sea por su función, por herencia, o por cualquier característica que deseemos, lo importante es que tengamos una organización de nuestras clases, ya que solo en la versión estándar de Java SE encontramos más de 4000 clases, y por lo tanto es importante definir la forma en que organizaremos nuestras clases.

Al tener tantas clases, es normal que existan clases con el mismo nombre, sin embargo como cada clase pertenece a un paquete distinto, los paquetes también nos servirán para evitar problemas de nombres entre clases.

CONVENCIÓN EN NOMBRE DE PAQUETES EN JAVA

- Todo el nombre debe estar en minúsculas.
- Se acostumbra escribir el nombre del dominio web de manera invertida.
Ejem. Si el nombre del dominio es: globalmentoring, entonces el nombre del paquete será: mx.com.globalmentoring
- Ejemplo de un proyecto mx.com.globalmentoring.miproyecto
- Ejemplo de subpaquetes
mx.com.globalmentoring.contabilidad.miproyecto
mx.com.globalmentoring.administracion.miproyecto

EJEMPLO DE CLASE DENTRO DE PAQUETE

Ejemplo de paquetes en Java:

```
package com.gm; //Definición del paquete

public class Utileria {
    public static void imprimir(String s){
        System.out.println("Imprimiendo mensaje: " + s);
    }
}

import com.gm.*; //Se importa el paquete a utilizar

public class EjemploPaquetes {

    public static void main(String[] args) {
        Utileria.imprimir("Hola"); //Se utiliza la clase importada
    }
}
```

Podemos observar en el ejemplo que primeramente creamos una clase llamada Utilería, esta clase la estamos agregando a un paquete llamado com.gm.

En este caso creamos la clase EjemploPaquetes en el paquete por default de Java, es decir, en ningún paquete.

Ahora, para poder utilizar la clase Utilería definida en el paquete com.gm, lo que debemos hacer es utilizar la palabra import, la cual la podemos utilizar de dos maneras, una es importando todas las clases usando el

*, o la otra es especificando el nombre de la clase, es decir: import com.gm.Utileria. Sin embargo, esta última opción nos obliga a hacer un import por cada clase que deseemos utilizar del paquete com.gm, y si

fueran muchas clases serían muchas líneas de código, por lo que en muchas ocasiones la notación import com.gm.* es la más utilizada. Cabe mencionar que el import no afecta la memoria, ya que no es que se carguen todas las clases de un paquete en memoria, una clase de carga en memoria hasta que se usa el nombre de la misma dentro del programa y no antes.

Finalmente podemos ver cómo estamos utilizando el método estático llamado imprimir, el cual pertenece a la clase Utilería. Observamos que ya no estamos indicando a qué paquete corresponde dicha clase, ya

que hemos hecho el import. Sin embargo, si hubiera más de una clase llamada Utilería, podríamos eliminar el import e indicar en la línea de código donde vamos a utilizar la clase, directamente el paquete al cual

corresponde la clase, quedando com.gm.Utileria.imprimir("Hola"); con esto sabríamos exactamente a qué paquete corresponde la clase que estamos utilizando.

IMPORT ESTÁTICO

Ejemplo de import estático en Java:

```
package com.gm; //Definición del paquete

public class Utileria {
    public static void imprimir(String s){
        System.out.println("Imprimiendo mensaje: " + s);
    }
}

//Importamos el método estático a utilizar
import static com.gm.Utileria.imprimir;

public class EjemploPaquetes {

    public static void main(String[] args) {
        imprimir("Hola");
    }
}
```

En el ejemplo mostrado podemos observar que la clase Utileria y el paquete al que pertenecen son los mismos que en el ejemplo anterior, pero debido a que el método imprimir es un método estático, podemos aprovechar la sintaxis de import static que nos brinda java para importar el método estático a utilizar, y de esta manera se simplifica la sintaxis del uso del método imprimir, ya que como podemos observar el método imprimir ya no debe indicar a qué clase pertenece, sino que es suficiente con indicar el nombre del método, ya que en el import static ya se ha indicado que pertenece a la clase Utileria.

22.- PALABRA FINAL EN JAVA

Usos de la palabra Final

- EN VARIABLES: Evita el valor que almacena la variable.

Es posible inicializar una variable final, pero una vez que se ha inicializado su valor, ya no es posible modificarlo, sin embargo recordemos que las variables que almacenan referencias de objetos no contienen el valor en si del objeto, sino solo de la referencia donde se ubica el mismo. Esto quiere decir que es posible cambiar el estado del objeto, pero no la referencia almacenada en la variable que se creó indicando que es final. En pocas palabras se dice que la palabra final es como crear una constante en Java, sin embargo debemos tomar en cuenta lo mencionado respecto a los objetos.

Debido a que la palabra final es como crear una constante, es muy común que se combine con la palabra static para poderla acceder directamente en lugar de crear una instancia de la clase para poder usar la variable final, en cambio al definir una variable como public static y final podemos accederla directamente indicando el nombre de la clase y posteriormente el nombre de la variable, y más aún, si agregamos el import static podemos utilizar directamente el nombre de la constante como veremos más adelante.

- EN MÉTODOS: Evita que se modifique la definición de un método desde una subclase

Cuanto estamos hablando de utilizar la palabra final en la definición de un método, básicamente estamos diciendo que una subclase no puede sobreescribir el comportamiento del método de la clase padre, es decir, que así como lo hereda debe utilizarlo y no puede modificarlo.

- EN CLASES: Evita que se cree una subclase

Y finalmente, cuando hablamos de clases, cuando definimos una clase como final, lo que indicamos es que no se puede crear clase que extienda de esta clase, es decir, no tendrá clases hijas.

```
//Clase Persona
public class Persona {
    private String nombre;
    public String getNombre() {
        return nombre;
    }

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

```
//Clase Clas Final
public static final class CalseFinal {

    //Variables marcadas como finales
    public static final int VAR_PRIMITIVO = 10;

    public static final Persona VAR_PERSONA = new Persona();

    //metodo final, lo estudiaremos a detalle en el tema de sobreescritura
    public final void metodoFinal(){

    }
}
```

```
//Calse Ejemplofinal
public class Ejemplofinal {
    public static void main(String[] args) {
        //Modificar un atributo final
        //CalseFinal.VAR_PRIMITIVO = 11;

        //Modificar la referencia de un atributo de tipo Object
        //CalseFinal.VAR_PERSONA = new Persona();

        CalseFinal.VAR_PERSONA.setNombre("Juan");
        System.out.println(CalseFinal.VAR_PERSONA.getNombre());

        CalseFinal.VAR_PERSONA.setNombre("Luis");
        System.out.println(CalseFinal.VAR_PERSONA.getNombre());
    }
}
```

23.- ARREGLOS EN JAVA



Por ejemplo, en la figura podemos observar dos arreglos, uno de tipo enteros y otro de tipos Persona. Es decir, que podemos declarar arreglos que contengan cualquier tipo de datos, ya sea de tipos primitivos o de tipo Object.

En el primer caso, el arreglo contiene 8 elementos (length), los cuales se numeran del elemento 0 al elemento 7. Esto quiere decir que los arreglos en Java inician en el índice 0, y el último elemento tendría el índice número de elementos menos uno.

Tenemos por otro lado el arreglo de tipo Persona, el cual contiene 7 elementos (length), y por ello el índice va de 0 a 6, según se observa en la figura.

No todos los elementos del arreglo deben contener valores. Por ejemplo, si el arreglo de enteros fuera de 10 elementos, pero solo tuviera 7 valores, los 3 últimos valores tendrían su valor por default del tipo declarado, en este caso como es de tipo int, el valor por default para el tipo int es 0.

En el caso del arreglo de tipo object si fuera de 10 elementos, y tuviera solo 5 objetos de tipo persona definidos, entonces los 5 restantes su valor serial null, ya que ese es el tipo por default para los tipo Object.

Declaración Arreglos

DECLARACIÓN ARREGLOS

- Sintaxis para declarar un arreglo de una dimensión:

tipo [] nombreArreglo	ó	tipo nombreArreglo [];
-----------------------	---	------------------------

- Ejemplo de declaración de arreglos de tipo primitivo:

int[] enteros;	ó	int enteros[];
boolean[] banderas;	ó	boolean banderas[];

- Ejemplo de declaración de arreglos de tipo Object:

Persona[] personas;	ó	Persona personas[];
String[] nombres;	ó	String nombres[];

CURSO DE FUNDAMENTOS DE JAVA

www.globalmentoring.com.mx

Declarar un arreglo es igual a declarar variables, podemos declarar arreglos que almacenen tipos primitivos o que almacenen referencias a objetos. En la lámina mostramos ambos casos, primero mostramos dos ejemplos de tipo primitivo, uno de tipo int y otro de tipo boolean. Posteriormente mostramos la declaración de dos arreglos que almacenarán referencias de objetos de tipo Persona y de tipo String.

JVM no sabe cuan largo es este arreglo, para ello debemos inicializarlo, veamos como.

Instanciar Arreglos

INSTANCIAR ARREGLOS

- Sintaxis para instanciar un arreglo de una dimensión:

```
nombreArreglo = new tipo[largo];
```

- Ejemplo para instanciar arreglos de tipo primitivo:

```
enteros = new int[10] ; //Arreglo de tipo entero de largo 10  
banderas = new boolean[5]; //Arreglo de tipo boolean de largo 5
```

- Ejemplo para instanciar arreglos de tipo Object:

```
personas = new Persona[13]; //Arreglo de tipo Persona de largo 13  
nombres = new String[8]; //Arreglo de tipo String de largo 8
```

Partiendo de las variables definidas en la lámina anterior, en esta lámina vemos la sintaxis para instanciar arreglos de una dimensión según el tipo de dato que estemos utilizando.

La sintaxis es muy similar a instanciar una variable de tipo object, y de hecho esta es una de las características de Java, incluso los arreglos o cualquier tipo en Java que almacena una referencia hereda de la clase Object de manera directa o indirecta, por lo tanto los arreglos también descienden de la clase Object.

Inicializar los elementos de un Arreglo

INICIALIZAR LOS ELEMENTOS DE UN ARREGLO

- Sintaxis para inicializar los elementos de un arreglo de una dimensión:

```
nombreArreglo[indice] = valor;
```

- Ejemplo para inicializar los elementos de un arreglo de tipo entero:

```
enteros[0] = 15; //Se asigna el valor de 15 en el indice 0  
enteros[1] = 13; //Se asigna el valor de 13 en el indice 1
```

- Ejemplo para inicializar los elementos de un arreglos de tipo Object:

```
personas[0] = new Persona(); //Se asigna el objeto persona en indice 0  
personas[1] = new Persona("Pedro","Lara"); //Se asigna objeto en indice 1  
nombres[0] = new String("Juan"); //Se asigna el String en indice 0  
nombres[1] = new String("Sara"); //Se asigna el String en indice 1
```

En la lámina podemos observar la inicialización de los elementos de un arreglo de una dimensión. Lo que debemos hacer es para ir agregando elementos a un arreglo, es seleccionar uno por uno los índices que queremos ir inicializando.

Por ello, es importante saber que el índice cero es el primer elemento del arreglo, y el último elemento de un arreglo lo podemos obtener con la propiedad length menos un elemento, por ejemplo si escribimos: enteros.length -1 nos devolverá el último índice del arreglo que podremos ocupar. Si nos pasamos del índice máximo y queremos agregar un elemento fuera de la cantidad máxima de elementos nos arrojará un error, por ello debemos saber cual es el máximo número de elementos con el código mencionado.

Extraer los elementos de un arreglo

EXTRAER ELEMENTOS DE UN ARREGLO

- Sintaxis para extraer los elementos de un arreglo de una dimensión:

```
variableReceptora = nombreArreglo[indice];
```

- Ejemplo para extraer los elementos de un arreglo de tipo entero:

```
int i = enteros[0]; //Extraemos el valor almacenado en el indice 0  
int j = enteros[1]; //Extraemos el valor almacenado en el indice 1
```

- Ejemplo para extraer los elementos de un arreglo de tipo Object:

```
Persona p1 = personas[0];//Extraemos valor almacenado indice 0  
Persona p2 = personas[1];//Extraemos valor almacenado indice 1  
String nombre1 = nombres[0]; //Extraemos valor almacenado indice 0  
String nombre2 = nombres[1]; //Extraemos valor almacenado indice 1
```

Para leer o extraer los elementos almacenados en un arreglo basta con indicar el nombre del arreglo e indicar el índice del elemento que queremos extraer, esto regresará el elemento del índice indicado.

Declaración, instanciación e inicialización

DECLARACIÓN, INSTANCIACIÓN E INICIALIZACIÓN

- Sintaxis para declarar, instanciar e inicializar los elementos de un arreglo:

```
tipo [] nombreArreglo = {lista de valores separados por coma};
```

- Ejemplo para declarar, instanciar e inicializar los elementos de un arreglo:

```
int[] edades = {10,23,41,68,7}; //arreglo de enteros de 5 elementos
```

- Ejemplo para declarar, instanciar e inicializar los elementos de un arreglo:

```
Persona[] personas = {new Persona(), new Persona("Juan","Perez")};  
String nombres = {"Karla","Arturo","Leandro","Sara"}; //4 elementos
```

Existe otra forma de declarar arreglos, y al mismo tiempo instanciarlo e inicializar cada uno de sus elementos. Esta es una sintaxis distinta en la forma en que se asignan los valores.

Sin embargo, esta sintaxis no siempre es posible utilizarla ya que necesitaríamos saber de antemano todos los elementos que van a ser almacenados en el arreglo, y en muchas ocasiones no tenemos esta información desde un inicio, pero si tenemos esta información antes de crear nuestro arreglo, entonces es posible utilizar esta sintaxis simplificada.

Ejemplo de manejo de arreglos

EJEMPLO DE MANEJO DE ARREGLOS

Ejemplo de arreglos de una dimensión:

```
1 public class EjemploArreglos {  
2     public static void main(String[] args) {  
3         //1. Declaramos un arreglo de enteros  
4         int edades[];  
5         //2. Instanciamos el arreglo de enteros  
6         edades = new int[3];  
7         //3. Inicializamos los valores del arreglo de enteros  
8         edades[0] = 30;  
9         edades[1] = 15;  
10  
11         //4. leemos los valores de cada elemento del arreglo  
12         System.out.println("Arreglo enteros indice 0: " + edades[0]);  
13         System.out.println("Arreglo enteros indice 1: " + edades[1]);  
14  
15         Persona personas[];  
16         personas = new Persona[4];  
17         personas[0] = new Persona("Juan");  
18         personas[1] = new Persona("Karla");  
19  
20         System.out.println("Arreglo personas indice 0: " + personas[0]);  
21         System.out.println("Arreglo personas indice 1: " + personas[1]);  
22     }  
23 }
```

Desde la declaración (líneas 5 y 16), el instanciamiento (líneas 7 y 17), la inicialización de valores (líneas 9-10 y 18-19), y finalmente la lectura de los valores (líneas 13-14 y 21-22).

Ejemplo recorrer un arreglo ciclo for

EJEMPLO RECORRER UN ARREGLO CICLO FOR

Ejemplo para recorrer un arreglo con un ciclo for:

```
1 public class EjemploArreglos {
2
3     public static void main(String[] args) {
4
5         //1. Arreglo de String, notación simplificada
6         String nombres[] = {"Sara", "Laura", "Carlos", "Carmen"};
7         //Imprimimos los valores a la salida estandar
8         //2. leemos los valores de cada elemento del arreglo
9         System.out.println("");
10        //Iteramos el arreglo de String con un for
11
12        for (int i = 0; i < nombres.length; i++) {
13            System.out.println("Arreglo String indice " + i + ": " + nombres[i]);
14        }
15    }
16 }
```

En primer lugar vemos un ejemplo del uso de la notación simplificada (línea

En este caso es un arreglo de tipo String, y en la misma línea instanciamos el arreglo e inicializamos los valores del arreglo. En este caso no hay que indicar el número de elementos que contendrá el arreglo, este número se obtendrá directamente del número de elementos que se agreguen en la inicialización del arreglo. Cabe aclarar que en esta estructura de datos no es posible hacer más grande o más pequeño el arreglo una vez declarado o como en este caso una vez inicializado.

24.- MATRICES EN JAVA

MATRICES EN JAVA							
Matriz de tipos int							
Primeros índices de la matriz		0	1	2	3	4	5
0	35	78	34	50	41	15	18
1	27	90	24	48	56	89	78
2	3	47	79	28	64	40	52
3	56	2	31	75	36	13	49

En la figura podemos observar una matriz de 4 renglones por 7 columnas, de tipo enteros, sin embargo puede ser de cualquier tipo que definamos.

Podemos recuperar el largo de los renglones con el código nombreArreglo.length y podemos obtener el largo de las columnas escribiendo nombreArreglo[0].length, es decir, que con cualquier renglón válido seleccionado podemos obtener el largo de las columnas.

Declaración

matriz

DECLARACIÓN MATRIZ

- Sintaxis para declarar una matriz:

```
tipo [][] nombreArreglo      ó      tipo nombreArreglo [][];
```

- Ejemplo de declaración de arreglos de tipo primitivo:

```
int[][] enteros;           ó          int enteros[][];  
boolean[][] banderas;       ó          boolean banderas[][];
```

- Ejemplo de declaración de arreglos de tipo Object:

```
Persona[][] personas;        ó          Persona personas[][];  
String[][][] nombres;        ó          String nombres[][][];
```

La sintaxis es muy similar a instanciar una variable de tipo object, y de hecho esta es una de las características de Java, incluso las matrices o cualquier tipo en Java que almacena

una referencia hereda de la clase Object de manera directa o indirecta, por lo tanto las matrices también descenden de la clase Object..

Instanciar Matrices

INSTANCIAR MATRICES

- Sintaxis para instanciar una matriz:

```
nombreArreglo = new tipo[renglones][columnas];
```

- Ejemplo para instanciar matrices de tipos primitivos:

```
enteros = new int[2][2] ;//Matriz tipo int: 2 renglones y 2 columnas  
anderas = new boolean[3][2];//Matriz de tipo boolean: 3 ren y 2 col
```

- Ejemplo para instanciar matrices de tipo Object:

```
personas = new Persona[4][2]; //Matriz tipo Person: 4 ren y 2 col  
nombres = new String[5][3]; //Matriz tipo String: 5 ren y 3 col
```

La sintaxis es muy similar a instanciar una variable de tipo object, y de hecho esta es una de las características de Java, incluso las matrices o cualquier tipo en Java que almacena una referencia hereda de la clase Object de manera directa o indirecta, por lo tanto las matrices también descenden de la clase Object.

Otra manera de Declarar e instanciar

```
tipo nombre del arreglo = new tipo [renglones] [columnas];
```

ejemplo

```
int edades = new int [3][2];
```

INICIALIZAR LOS ELEMENTOS DE UNA MATRIZ

- Sintaxis para inicializar los elementos de una matriz:

```
nombreArreglo[indice_renglon][indice_columna] = valor;
```

- Ejemplo para inicializar los elementos de una matriz de tipo entero:

```
enteros[0][0] = 15; //Se asigna el valor de 15 en el ren=0 y col=0  
enteros[1][0] = 13; //Se asigna el valor de 13 en el ren=1 y col=0
```

- Ejemplo para inicializar los elementos de un arreglos de tipo Object:

```
personas[0][0] = new Persona(); //Se asigna objeto en ren=0 y col=0  
personas[1][1] = new Persona("Pedro","Lara"); //Se asigna en ren=1 y col=1  
nombres[0][0] = new String("Juan"); //Se asigna String en ren=0, col=0  
nombres[2][1] = new String("Sara"); //Se asigna String en ren=2, col=1
```

En la lámina podemos observar la inicialización de los elementos de una matriz. Lo que debemos hacer para ir agregando elementos a una matriz, es seleccionar un renglón y una columna con los índices respectivos que queremos ir inicializando.

Por ello, es importante saber que a diferencia de un arreglo, en una matriz utilizaremos dos índices para determinar la posición de un elemento, y que los primeros índices tanto del renglón como de la columna inician en cero. También es importante saber que cuando indicamos una posición primero se indica el renglón y después la columna, siempre en ese orden. Por ejemplo, el primer elemento de una matriz será el elemento [0][0] y el largo de una matriz realmente son dos, el primero lo determinaremos por el nombreMatriz.length lo que nos regresa el largo de renglones, y posteriormente podemos saber el largo de las columnas seleccionando cualquier renglón, por ejemplo: nombreMatriz[0].length.

Al igual que en un arreglo, sólo podemos agregar elementos hasta el máximo de elementos menos uno, por ejemplo, si son renglones, sería nombreMatriz.length -1 y si fuera el máximo de columnas sería nombreMatriz[i].length -1, donde i es el renglón que se está trabajando. Si nos pasamos del índice máximo tanto en renglones o columnas y queremos agregar un elemento fuera de la cantidad máxima de elementos nos arrojará un error, por ello debemos saber cual es el máximo número de elementos tanto en renglones como en columnas.

Podemos observar en la lámina varios ejemplos de cómo agregar elementos a nuestra matriz. Podemos agregarlos de manera manual, es decir, uno a uno cada elemento, o podemos ir agregando los elementos de manera más dinámica utilizando dos contadores de elementos que han sido agregados tanto para los renglones como para las columnas, de tal forma que podamos saber si ya hemos llegado al límite de elementos agregados o no

EXTRAER ELEMENTOS DE UNA MATRIZ

- Sintaxis para extraer los elementos de una matriz:

```
variableReceptora = nombreArreglo[indice_renglón][indice_columna];
```

- Ejemplo para extraer los elementos de una matriz de tipo entero:

```
int i = enteros[0][0]; //Extraemos valor almacenado en ren 0 y col 0
int j = enteros[1][0]; //Extraemos valor almacenado en ren 1 y col 0
```

- Ejemplo para extraer los elementos de una matriz de tipo Object:

```
Persona p1 = personas[0][0];//Extraemos valor de ren 0 y col 0
Persona p2 = personas[1][0];//Extraemos valor de ren 1 y col 0
String nombre1 = nombres[0][0]; //Extraemos valor de ren 0 y col 0
String nombre2 = nombres[1][1]; //Extraemos valor de ren 1 y col 1
```

DECLARACIÓN, INSTANCIACIÓN E INICIALIZACIÓN

- Sintaxis para declarar, instanciar e inicializar los elementos de una matriz:

```
    tipo [][] nombreArreglo = { lista_valores }, { lista_valores };
```

- Ejemplo para declarar, instanciar e inicializar los elementos de un arreglo:

```
        columnas
int[][] edades = { 10, 23, 41 }, { 10, 23, 41 }, { 10, 23, 41 }, { 10, 23, 41 };
```

- Ejemplo para declarar, instanciar e inicializar los elementos de un arreglo:

```
Persona[][] personas = { new Persona(), new Persona() }, { new Persona(), new Persona() };
String nombres = { "Karla", "Arturo", "Juan" }, { "Pedro", "Laura", "Oscar" };
```

Ejemplo de manejo de matrices

EJEMPLO DE MANEJO DE MATRICES

Ejemplo de uso de matrices:

```
1 public class EjemploMatrices {
2
3     public static void main(String[] args) {
4         //1. Declaramos un arreglo de enteros
5         int edades[][];
6         //2. Instanciamos el arreglo de enteros
7         edades = new int[3][2];
8         //3. Inicializamos los valores del arreglo de enteros
9         edades[0][0] = 30;
10        edades[0][1] = 15;
11        edades[1][0] = 20;
12        edades[1][1] = 45;
13        edades[2][0] = 5;
14        edades[2][1] = 38;
15
16        //Imprimimos los valores a la salida estándar
17        //4. leemos los valores de cada elemento del arreglo
18        System.out.println("Arreglo enteros indice 0-0: " + edades[0][0]);
19        System.out.println("Arreglo enteros indice 0-1: " + edades[0][1]);
20        System.out.println("Arreglo enteros indice 1-0: " + edades[1][0]);
21        System.out.println("Arreglo enteros indice 1-1: " + edades[1][1]);
22        System.out.println("Arreglo enteros indice 2-0: " + edades[2][0]);
23        System.out.println("Arreglo enteros indice 2-1: " + edades[2][1]);
24    }
25 }
```

Desde la declaración (línea 5), el instanciamiento (línea 7), la inicialización de valores (líneas 9-14), y finalmente la lectura de los valores (líneas 18-23).

EJEMPLO RECORRER UNA MATRIZ CICLO FOR ANIDADO

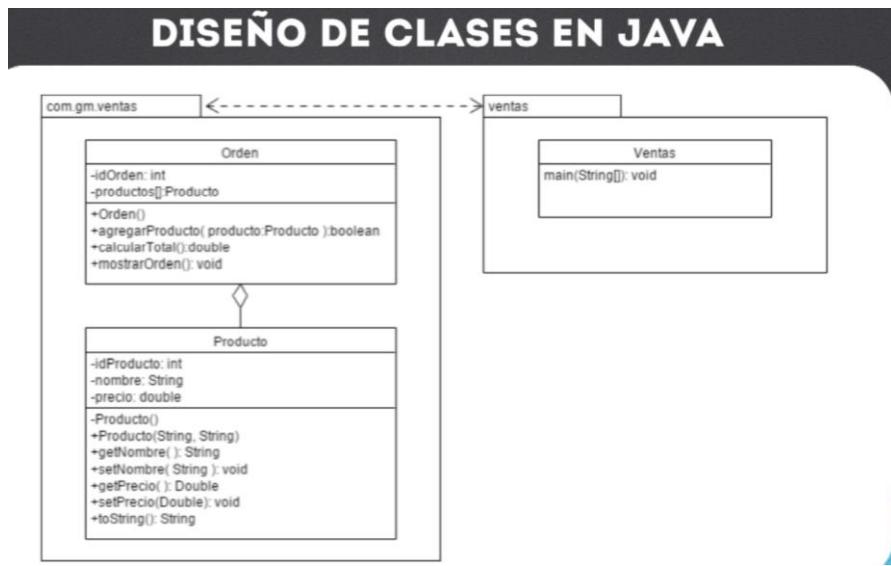
Ejemplo para recorrer una matriz con un ciclo for anidado:

```
1 public class EjemploMatrices {  
2  
3     public static void main(String[] args) {  
4  
5         //1. Matriz de tipo String, notación simplificada  
6         String nombres[][] = {{"Teresa", "Cesar", "William"}, {"Yesenia", "Esteban", "Maria"};  
7  
8         //Largo de elementos de la matriz. Primero el no. de renglones  
9         System.out.println("largo matriz renglones:" + nombres.length);  
10        //Seleccionando un renglón válido nos regresa el no. de columnas  
11        System.out.println("largo matriz columnas:" + nombres[0].length);  
12  
13        //Imprimimos los valores a la salida estándar  
14        //2. Iteraremos la matriz de String con un for anidado  
15        for (int i = 0; i < nombres.length; i++) {  
16            for(int j= 0; j < nombres[i].length; j++){  
17                System.out.println("Matriz String indice : " + i + "-" + j + " : " + nombres[i][j]);  
18            }  
19        }  
20    }  
21 }
```

```
Output - EjemploMatrices (run) X |  
run:  
largo matriz renglones:2  
largo matriz columnas:3  
  
Matriz String indice : 0-0: Teresa  
Matriz String indice : 0-1: Cesar  
Matriz String indice : 0-2: William  
Matriz String indice : 1-0: Yesenia  
Matriz String indice : 1-1: Esteban  
Matriz String indice : 1-2: Maria
```

El primer ciclo, el más externo recorre los renglones de la matriz, y el ciclo más interno recorre las columnas de la matriz. Por ello en la salida de nuestra consola, observaremos que los primeros 3 valores el valor del índice del renglón se mantiene fijo, mientras que índice de la columna se va moviendo hasta que se acaba de iterar las columnas para ese renglón seleccionado. El fin del ciclo más interno será cuando hayamos iterado todas las columnas para el renglón seleccionado, según la condición del ciclo for interno: $j < \text{nombres}[i].length$ (línea 16). Recordemos que la variable i controla los renglones, y la variable j controla las columnas. Finalmente el ciclo más externo se detendrá cuando se hayan revisado todos los renglones de la matriz, según la condición del ciclo for externo: $i < \text{nombres.length}$ (línea 15). Con esto habremos iterado todos los renglones, así como cada columna de cada renglón seleccionado, y por consiguiente todos los elementos de la matriz.

25.- DISEÑO DE CLASE EN JAVA



Las clases están relacionadas por una relación en UML (Unified Modeling Language) que se conoce como relación de agregación. Esta relación indica que una clase Orden contiene Productos. Una orden es un ticket de venta, el cual tiene el resumen de todos los productos que se van a vender para una orden en particular.

Por ello la clase Orden contiene los métodos de agregarProducto, calcularTotal (de la orden), y mostrarOrden, este último método su objetivo es mostrar el Id de la orden, el monto total de la orden, así como cada uno de los productos agregados a la orden. Para almacenar varios productos, la clase Orden tiene como atributo un arreglo de productos, y de esta manera podremos agregar varios productos a una orden, con ayuda del método agregarProducto.

La clase Producto contiene tres atributos: idProducto, nombre y precio del producto, por medio de estos atributos podemos identificar fácilmente a un producto.

Finalmente tenemos la clase con la que realizaremos las pruebas de que todo funciona correctamente. La clase Ventas dentro del paquete ventas, es donde crearemos los objetos Orden y Producto y utilizaremos los métodos respectivos para probar que funciona correctamente nuestro código.

```

// Clase Productos

package com.gm.ventas;

public class Producto {
    private int idProducto;
    private String nombre;
    private double precio;
    private static int contadorProductos;
}

//Agregamos un constructor vacio
private Producto (){

    this.idProducto=
    ++contadorProductos;
}

//Constructor sobrecargado de
argumentos
public Producto(String nombre, double
precio){
    this();
    this.nombre =nombre ;
    this.precio = precio;
}

public int getIdProducto() {
    return idProducto;
}

public void setIdProducto(int
idProducto) {
    this.idProducto = idProducto;
}

public String getNombre() {
    return nombre;
}

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

public double getPrecio() {
    return precio;
}

public void setPrecio(double precio) {
    this.precio = precio;
}

@Override
public String toString() {
    return "Producto{" +
        "idProducto=" + idProducto +
        ", nombre='" + nombre + '\'' +
        ", precio=" + precio +
        '}';
}

```

```

//Clase Orden
package com.gm.ventas;

public class Orden {
    private int idOrden;
    private Producto[] productos;
    private static int contadorOrdenes;
    private int contadorProductos;
    private static final int MAX_PRODUCTOS = 10;

    public Orden() {
        this.idOrden = ++contadorOrdenes;
        //inicializar el arreglo de objeto
        productos = new Producto[MAX_PRODUCTOS];
    }

    public void agregarproducto(Producto producto) {
        //Si los productos agregados no
        superan el maximo de productos
        //agregamos el nuevo producto al
        arreglo
        if (contadorProductos <
MAX_PRODUCTOS) {
            //agregamos el nuevo producto al
            arreglo
            // e incrementar el contador de
            productos
            productos[contadorProductos++] =
            producto;
        } else {
            System.out.println("Se ha
superado el maximo de productos: " +
MAX_PRODUCTOS);
        }
    }

    public double calcularTotal() {
        double total = 0;
        for (int i = 0; i < contadorProductos;
i++) {
            total += productos[i].getPrecio();
            //total = total + productos[i].getPrecio();
        }
        return total;
    }

    public void mostrarOrden(){
        System.out.println("Orden #: "
+idOrden);
        System.out.println("Total de la
orden: " +calcularTotal());
        for (int i =0; i< contadorProductos;
i++){
            System.out.println(productos[i]);
        }
    }
}

```

```
//Clase Ventas
package ventas;
import com.gm.ventas.*;
public class Ventas {
    public static void main(String[] args){
        //Creamos varios objetos de tipo productos
        Producto producto1 = new Producto("Camisa", 50);
        Producto producto2 = new Producto("Patalon", 150);

        //Creamos un objeto de tipo orden
        Orden orden1 = new Orden();
        Orden orden2 = new Orden();

        //Agregamos los productos a la orden
        orden1.agregarproducto(producto1);
        orden1.agregarproducto(producto2);
        orden1.agregarproducto(producto2);

        orden2.agregarproducto(producto2);
        orden2.agregarproducto(producto2);

        //Imprimir la orden
        orden1.mostrarOrden();
        System.out.println();
        orden2.mostrarOrden();

    }
}

}
```