

# IPOO

## ▶ Docentes

### ▶ Profesores Adjuntos:

▶ Carlos Di Cicco.

### ▶ JTPs:

▶ Federico Naso (Junín) .  
Nelson Di Grazia (Pergamino) .

# Objetos y clases

- ▶ Una instancia u **objeto es una entidad de software** que combina un estado/datos y comportamiento/métodos.
- ▶ Una **clase es un bloque de código o molde, que describe cómo son los objetos que pertenecen a ella**. Contiene variables que representan los estados de los objetos y métodos que representan los mensajes que entienden los objetos a través de los cuales definen su comportamiento.
- ▶ Cada instancia de una clase (objeto) tiene una copia de las variables de instancia y de los métodos declarados en la clase.

# Objetos y clases

## Componentes en la declaración de la clase

Declaración	public	la clase es accesible públicamente
	abstract	la clase no puede instanciarse
	final	la clase no puede subclasearse
	<b>class <i>NombreDeClase</i></b>	<b>nombre de la clase (obligatorio)</b>
	extends Super	Super es la superclase de la clase
	implements <i>Interfaces</i>	interfaces implementadas por la clase
Cuerpo	{ Cuerpo de la Clase }	Constructores para la inicialización. Declaraciones de variables. Métodos que dan comportamiento.

Los valores por default, asumidos por el compilador JAVA son: no public, no abstract, no final, subclase de **Object** y no se implementan interfaces.

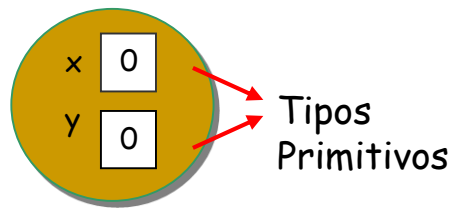
# Objetos y clases

```
public class Punto {  
    public int y = 0;  
    public int x = 0;  
    .....  
}
```

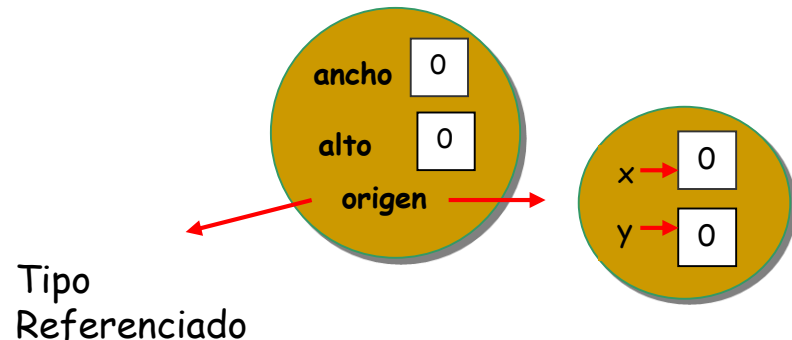
```
public class Rectangulo {  
    public int ancho = 0;  
    public int alto = 0;  
    public Punto origen = new Punto();  
    .....  
}
```

- Instanciación de una clase

new Punto();



new Rectangulo();



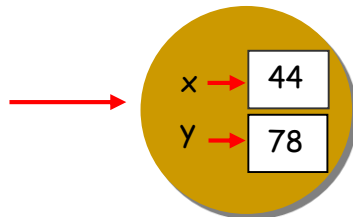
# Objetos y clases

```
public class Punto {  
    public int y = 0;  
    public int x = 0;
```

Inicialización  
de una clase

```
    public Punto (int x, int y)  
    {  
        this.x = x;  
        this.y = y;  
    }  
}
```

new Punto(44,78);



**this:** referencia al objeto corriente  
**super:** referencia a la superclase

3 Constructores

```
public class Rectangulo {  
    public int ancho=0;  
    public int alto=0;  
    public Punto origen;
```

```
    public Rectangulo() {  
        origen=new Punto(0,0);  
    }
```

```
    public Rectangulo(Punto p) {  
        origen=p;
```

```
    }  
    public Rectangulo(int w, int h) {  
        this (new Punto(0,0),w,h);  
    }
```

```
    public void mover(int x, int y) {  
        origen.x=x;  
        origen.y=y;
```

```
    }  
    public int area(int x, int y) {  
        return ancho*alto;  
    }  
}
```

Métodos de instancia

# Objetos y clases

Declaración de la clase

Cuerpo de la clase

```
package modelo;

public class Cuenta{
    private int cuentaId;
    private double saldo;
    private char tipo;

    public double getSaldo(){
        // código del método
    }
    public void setSaldo(double saldo){
        this.saldo = saldo;
    }
}
```

Indica el paquete donde se ubica la clase

Indica el comienzo de la clase

Variables de instancia

Métodos de instancia

Indica el final de la clase


# ¿Cómo incorporar estado y comportamiento a una clase?

- ▶ Se debe agregar en el cuerpo de la Clase misma:
- ▶ - **variables de instancia**: constituyen el estado de un objeto. Normalmente, las variables de instancia se declaran **private**, lo que significa que sólo la clase puede acceder a ellas, directamente.
- ▶ - **métodos de instancia**: definen las operaciones que pueden realizar los objetos de un tipo de clase. Un método es un bloque de código, similar a lo que es una función o procedimiento en los lenguajes procedurales.

# ¿Cómo incorporar estado y comportamiento a una clase?

La declaración de una **variables de instancia** debe incluir:

- Un identificador (nombre de la variable).
- Un tipo (tipo primitivo o de un tipo de una clase).
- Un modificador de acceso (opcional): **public** o **private**

```
public class Cliente {  
  
    private int clienteId;  declaración  
    private String domicilio;  
    private double deuda;  
  
    public double getDeuda() {  
    }  
  
    public void setDeuda(double d) {  
        deuda = d;  
    }  
  
    // más métodos de instancia  
}
```

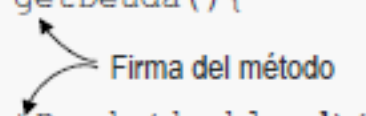


# ¿Cómo incorporar estado y comportamiento a una clase?

La declaración de un **método de instancia** debe especificar:

- Un nombre
- Una lista de argumentos (opcional)
- Un tipo de retorno
- Un modificador de acceso (opcional):  
**public** o **private**

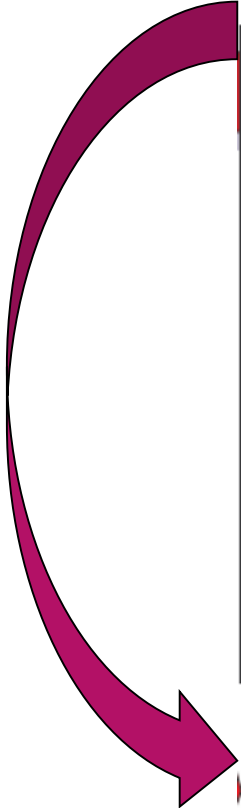
```
public class Cliente {  
  
    private int clienteId;  
    private String domicilio;  
    private double deuda;  
  
    public double getDeuda() {  
    }  
  
    public void setDeuda(double d) {  
        deuda = d;  
    }  
  
    // más métodos de instancia  
}
```













The diagram shows two arrows originating from the text "Firma del método". One arrow points to the `public double` part of the `getDeuda()` method signature. The other arrow points to the `public void` part of the `setDeuda(double d)` method signature.

# Variables.

## Componentes en la declaración.



public protected package private	controla el nivel de acceso para esta variable
static	declara una variable de clase
final	declara una constante. El nombre de la misma debe estar en mayúscula.
transient	declara variables que no deben ser serializables. Se usan para denominar atributos que no forman parte del estado permanente de un objeto
volatile	este modificador evita que el compilador realice ciertas optimizaciones
<b>type name</b>	indica el tipo y el nombre de la variable. <b>Primitivos</b> (int, float, boolean) o <b>Referenciados</b> (array, object o nombres de interfaces)

Especificador	Clase	SubClase	Package	Todos
Private				
protected				
public				
package				

# Variables

- **Variables de instancia:** son las variables que se declaran en el cuerpo de la clase y afuera de los métodos

```
private String nombre;  
private char sexo= 'M';
```

- **Variables locales:** son las variables que se declaran adentro los métodos; deben inicializarse SIEMPRE

```
int contador = 0;
```

```
int contador;  
contador = 0;
```

- Toda variable **debe declararse** antes de usarse

# Variables. Nombrado

Toda variable debe tener un nombre, el cual se recomienda que sea simple pero descriptivo.

Un nombre debe comenzar con:

- ▶ Una letra minúscula
- ▶ Un guión bajo (\_)
- ▶ El signo dólar (\$)
- ▶ Después del primer carácter, se pueden usar dígitos

Un nombre no puede contener signos de puntuación (. , ? ¿ ! ;), ni espacios ( ), ni guión medio (-).

Por convención comienza con una letra minúscula.

# Variables. Nombrado

Las siguientes palabras reservadas no pueden ser usadas como identificadores

abstract  
assert  
boolean  
break  
byte  
case  
catch  
char  
class  
const  
continue

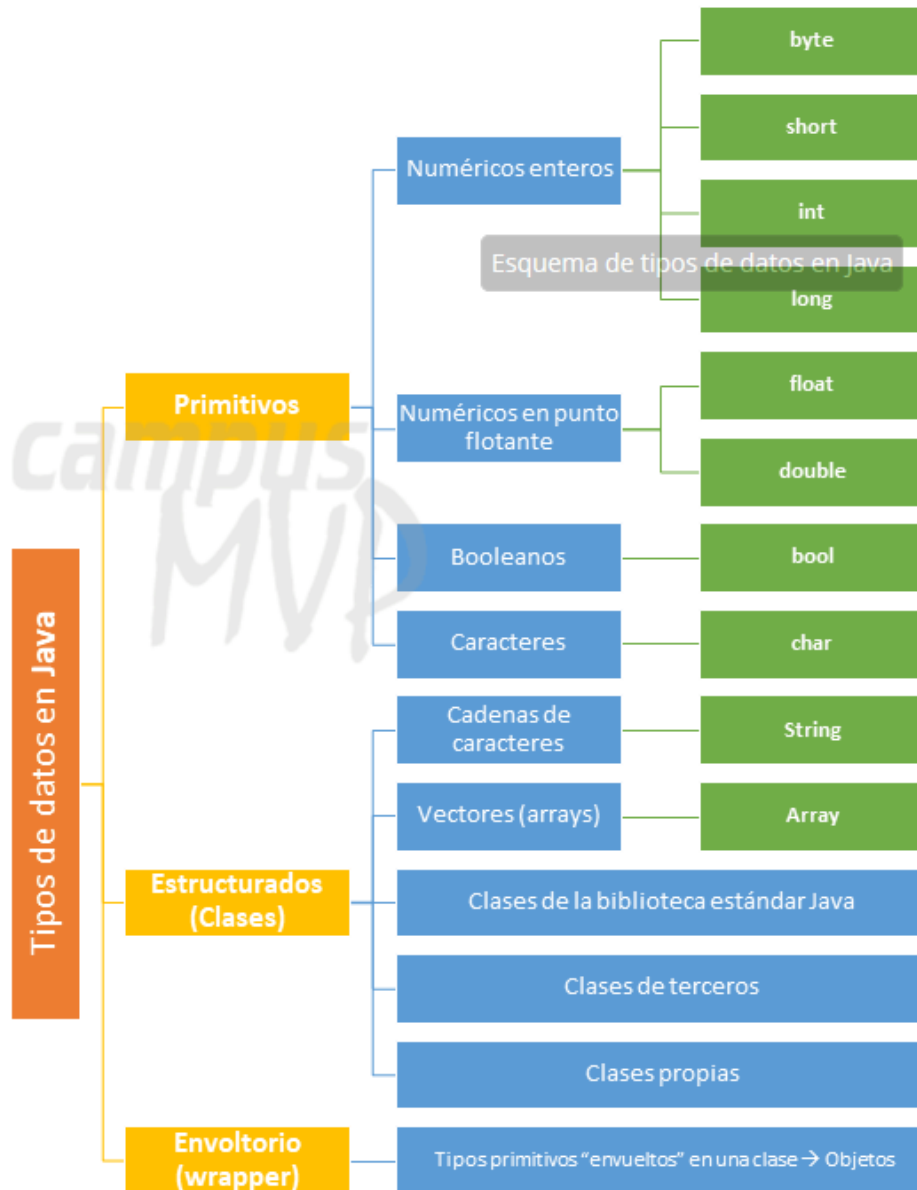
default  
do  
double  
else  
extends  
false  
final  
finally  
float  
for  
goto

if  
implements  
import  
instanceOf  
int  
interface  
long  
native  
new  
null  
package

private  
protected  
public  
return  
short  
static  
strictfp  
super  
switch  
synchronized  
this

throw  
throws  
transient  
true  
try  
void  
volatile  
while

# Tipos de datos



# Tipos de datos

En java hay 2 categorías de tipos de datos: tipo primitivo y tipo de una clase (referencia).

**Tipos primitivos:** las variables de tipo primitivo mantienen valores simples y NO son objetos. Existen 8 tipos de datos primitivos:

- Tipos Enteros: *byte, short, int, long*
- Tipos de Punto Flotante: *float, double*
- Tipo Textual: *char*
- Tipo Lógico: *boolean*

## Declaración e inicialización de variables de tipo primitivo

```
float pi = 3.14;  
double saldo = 0;  
char letra = 'F';  
int hora = 12;  
boolean es_am = (hora>12);
```

# Tipos de datos

Si la definición de una clase, no inicializa variables de instancia, las mismas toman valores por defecto.

- Las variables de instancia de tipo primitivo se inicializan con los siguientes valores por defecto:

Tipo primitivo	Valor por defecto
<code>boolean</code>	<code>false</code>
<code>char</code>	<code>'\u0000'</code> (nulo)
<code>byte/short/int/long</code>	<code>0</code>
<code>float/double</code>	<code>0.0</code>

- Las variables de instancia que son referencias a objetos, se inicializan con el valor por defecto: *null*.
- Las variables locales, es decir, las variables declaradas dentro de un método, deben inicializarse explícitamente antes de usarse.



# Variables. Tipos Enteros

Tipo	Longitud	Rango	Cantidad de valores
byte	8 bits	$-2^7$ a $2^7 - 1$ -128 a 127	256
short	16 bits	$-2^{15}$ a $2^{15} - 1$ -32.768 a 32.767	65.535
int	32 bits	$-2^{31}$ a $2^{31} - 1$ -2.147.483.648 a 2.147.483.647	4.294.967.296
long	64 bits	$-2^{63}$ a $2^{63} - 1$ -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807	18.446.744.073.709.551.61 6

Nota: cuando se especifica un valor para un tipo *long*, se debe escribir una “L” a la derecha del valor.  
No sucede lo mismo con los otros tipos.

# Variables. Tipos de Punto Flotante

Tipo	Longitud	Ejemplos
float	32 bits	99F -32745699,01F 4,2E6F
double	64 bits	-1111 2,1E12 99970132745699,999

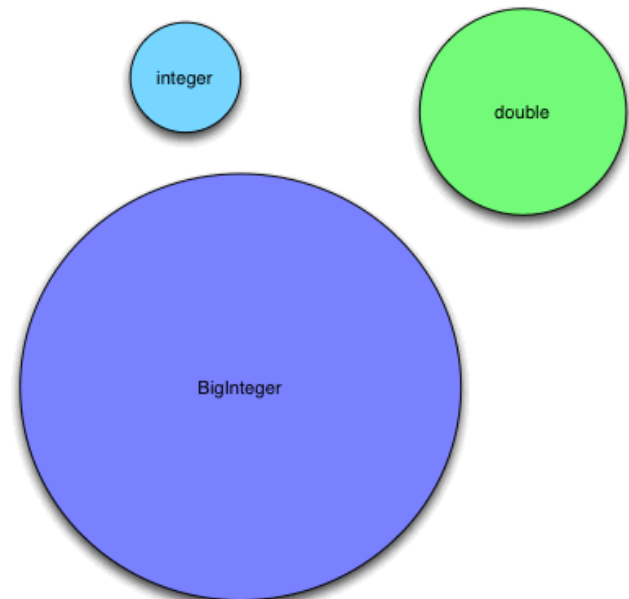
Observación: cuando se especifica un valor para un tipo float, se debe escribir una “F” a la derecha del valor. No sucede lo mismo con los valores double

# Tipos de datos grandes. BigInteger

En muchos cálculos, incluso comunes como el factorial, es necesario tener números grandes para almacenar el resultado.

La solución es una clase Java especial reservada para este tipo de operaciones y que se denomina **BigInteger**.

De esta forma podremos calcular el factorial de números mucho más grandes.



# Tipos de datos grandes. BigInteger

```
package com.arquitectura;

import java.math.BigInteger;

public class CalcularFactorial2 {

    public static void main(String[] args) {

        System.out.println(factorial(new BigInteger("1000")));

    }

    static BigInteger factorial(BigInteger n)
    {
        if (n.equals(BigInteger.ZERO))
            return BigInteger.ONE;
        else
            return n.multiply(factorial(n.subtract(BigInteger.ONE)));
    }

}
```

# BigDecimal

- Se utiliza para cálculos aritméticos financieros. Ejemplo:

Escribe el siguiente código:

```
double unCentavo = 0.01;  
double suma=unCentavo+unCentavo+unCentavo+unCentavo+unCentavo+unCentavo;  
System.out.println(suma);
```

Que se imprime? Si contestaste: **0.06**, estas equivocado.

Se imprime: **0.060000000000000005**

Ahora escribe:

```
java.math.BigDecimal unCentavo = new java.math.BigDecimal("0.01");  
java.math.BigDecimal  
suma=unCentavo.add(unCentavo).add(unCentavo).add(unCentavo).add(unCentavo).add(unCentavo);  
System.out.println(suma);
```

Que imprime? **0.06**.

# Variables. Tipo Lógico

Solo pueden almacenar:

- Los literales: *true* o *false*
- El resultado de una expresión que solo evalúa a *true* o *false*.

```
int respuesta = 42;  
if (respuesta < 42){...}
```

Evalúa a *false*

# Métodos

Componentes de la declaración de métodos.

Declaración	public protected package private	controla el nivel de acceso para el método	Componente obligatorio
	static	método de clase	
	abstract	método no implementado	
	final	método no puede ser sobrescrito por las subclases	
	native	método implementado en otro lenguaje	
	synchronized	método que requiere de un monitor para ejecutarse	
	<b>tipo_retornado NombreDelMetodo</b>	devuelve un objeto del tipo tipo_retornado. Los tipos pueden ser: <b>Primitivos</b> (int, float, boolean) o <b>Referenciados</b> (array, object o nombres de interfaces)	
	(lista de parámetros)	lista de argumentos ( <b>tipo nombre</b> ) separados por coma. Los tipos pueden ser: <b>Primitivos</b> (int, float, boolean) o <b>Referenciados</b> (array, object o nombres de interfaces)	
Cuerpo	throws exceptions	excepciones manejadas por el método	
	{ Cuerpo del Método }		

# Métodos

Los **métodos** de una clase se escriben al nivel de la declaración de los atributos de la clase.

## Declaración del método

```
public double getSaldo(double saldo) {  
    this.saldo = saldo;  
    return this.getSaldo();  
}
```

Tipo de retorno. Si el método no devuelve ningún valor, se utiliza la palabra **void**.

Nombre del método

Alcance del método

Cuerpo del método

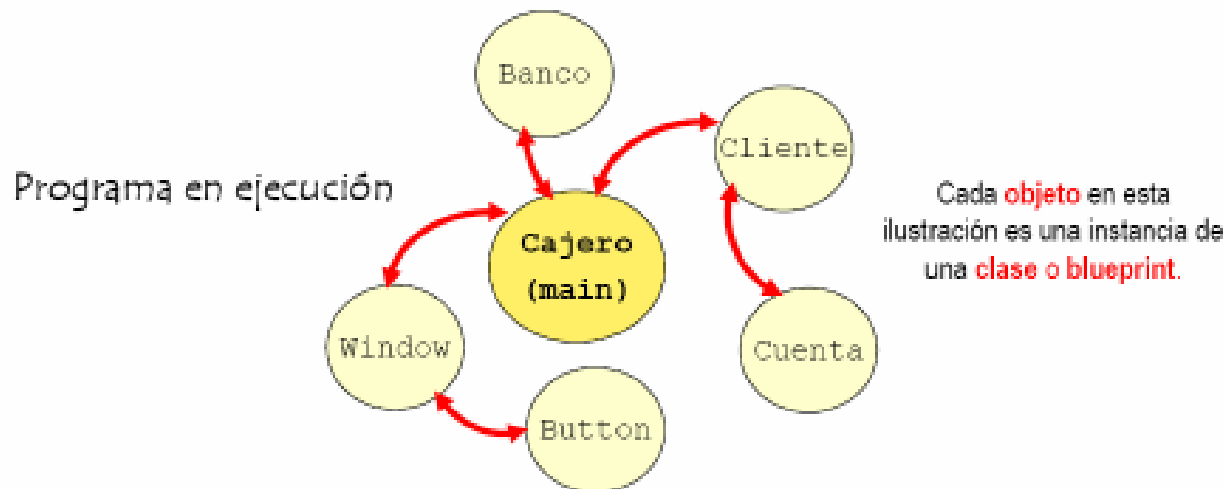
Argumentos indicando su tipo y su nombre.

cláusula de retorno.



# Método main

Una aplicación de escritorio, normalmente consiste de una clase principal, que es el punto de entrada de cada aplicación. Esta clase debe tener el método **main**.



Si se trata de correr una clase que no tiene el método **main**, el intérprete no podrá ejecutar la aplicación y disparará un error.

100

# Método main

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

calificadores obligatorios

Acepta objetos de tipo String, aunque no es obligatorio enviarlos.

Acepta objetos de tipo String, aunque no es obligatorio enviarlos.

Dada una clase Cajero, que puede ejecutarse:

```
public class Cajero {  
  
    public static void main(String[] args) {  
        System.out.println("Buen dia !!!"+args[0]+" "+args[1]);  
    }  
}
```

```
public class Cajero {  
  
    public static void main(String[] args) {  
        System.out.println("Buen dia !!!"+args[0]+" "+args[1]);  
    }  
}
```

Si ejecutamos desde la línea de comandos la clase Cajero:

`C:\java` **Cajero** María Jeréz → Buen día María Jeréz

argumentos salida



salida

# Constructores

- Un constructor sirve para inicializar los atributos o estados de un objeto.
- Si una clase NO declara constructores, el compilador inserta automáticamente un constructor nulo. Si la clase declara al menos un constructor, con o sin argumentos, el compilador NO insertará el constructor nulo.
- Una clase puede tener más de un constructor: **constructores sobrecargados**.

## Caso 1: Clase con constructores sobrecargados

```
public class Vehiculo {  
    private String nroPatente;  
    private String propietario;  
    public Vehiculo(String patente){  
        this.nroPatente = patente;  
    }  
    public Vehiculo(String patente, String propietario){  
        this(patente);  
        this.propietario= propietario;  
    }  
    public Vehiculo(){ }  
    // métodos  
}
```

## Caso 2: Clase que NO declara un constructor

```
public class Vehiculo {  
    private String nroPatente="";  
    private String propietario="SinDueño";  
    // El compilador agrega  
    // métodos ← - - - - - public Vehiculo(){ }  
}
```

# Constructores

- ▶ La creación e inicialización de un objeto involucra los siguientes pasos:

**Vehiculo v= new Vehiculo(“DWL120”, “Juan García”);**

- ▶ Alocación de espacio en memoria para la variable **v** y para el objeto **Vehiculo**.
- ▶ Inicialización de las variables de instancia del objeto con los valores por defecto de acuerdo al tipo de dato.
- ▶ Re-seteo de las variables de instancia con el valor definido en la declaración (si estos fueron definidos).
- ▶ Ejecución del constructor. Re-seteo de las variables de instancia de acuerdo al código del constructor.
- ▶ Asignación a la variable **v** de la referencia del nuevo objeto.

# Constructores

```
public class Vehiculo {  
    private String nroPatente="";  
    private String propietario="SinDueño";  
    public Vehiculo(String patente){  
        this.nroPatente = patente;  
    }  
    public Vehiculo(String patente, String propietario){  
        this(patente)  
        this.propietario= propietario;  
    }  
    public Vehiculo(){  
        // métodos  
    }  
}
```

Declaro e inicializo  
las variables de  
instancia en la  
misma línea

**v = 0x99f311**

**nroPatente= "AAA 123"**  
**propietario= "SinDueño"**

0x99f311

Si ejecutamos el siguiente código:

**Vehiculo v = new Vehiculo("AAA 123");**

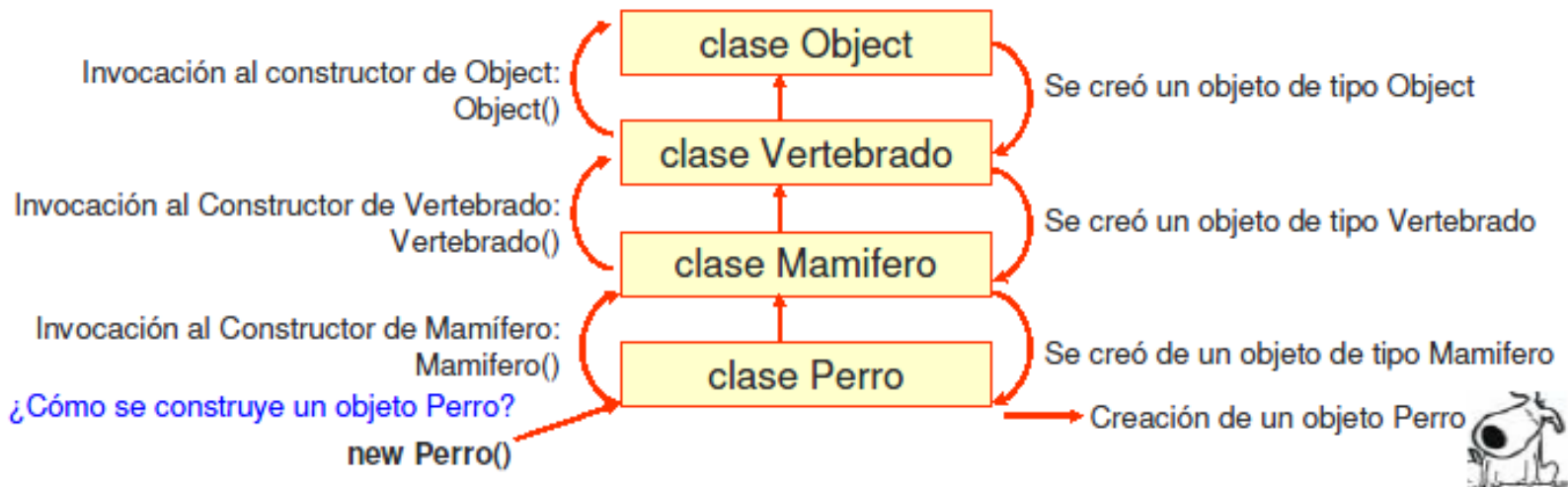
**¿Cómo es el proceso de creación del objeto?**

- 1) Alocación memoria para la variable **v** y el objeto **Vehiculo**.
- 2) Inicialización de las variables de instancia. ¿Qué valores tienen **nroPatente** y **propietario**?
- 3) Re-seteo de las variables con los valores de la declaración. ¿Qué valores toman **nroPatente** y **propietario**?
- 4) Ejecución del constructor y re-seteo de las variables de instancia. ¿Qué valores toman **nroPatente** y **propietario**?
- 5) Asignación a la variable **v** de la referencia al objeto  
¿Qué valor toma **v**?

# Constructores

## ¿Cómo se construye un objeto?

Recorriendo la jerarquía de herencia en forma ascendente e invocando al constructor de la superclase desde cada constructor en cada nivel de la jerarquía de clases:



En el constructor de cada clase existe una invocación al constructor de la superclase. Cada objeto contiene una referencia a un objeto de la superclase y ésta se crea en la invocación al constructor de la superclase. Mediante esta referencia, el objeto puede acceder a los métodos y variables de instancia de sus superclases.

# Constructores

## Ejemplo:

Disponemos de la clase **Vehiculo** y de su subclase **Automovil**. A su vez, **Automovil** define un atributo nuevo de tipo **int** llamado **cantidadPuertas**

```
public class Vehiculo {
    private String nroPatente="";
    private String propietario="SinDueño";
    public Vehiculo(String patente){
        this.nroPatente = patente;
    }
    public Vehiculo(String patente, String propietario){
        this.nroPatente = patente;
        this.propietario= propietario;
    }
    public Vehiculo(){
        // métodos
    }
}
```

```
public class Automovil extends Vehiculo {
    private int cantidadPuertas=4;
    public Automovil(String patente, String propietario, int puertas){
        super(patente,propietario);
        this.cantidadPuertas = puertas;
    }
    // métodos
}
```

Si en el constructor de **Automovil** quiero invocar al constructor de **Vehiculo** con los argumentos **patente** y **propietario** ¿Cómo lo hago?

Utilizo el **super()**. Este método es similar al **this()**, pero en lugar de invocar a un constructor de la misma clase, invoca a un constructor de la superclase. ¿Qué líneas de código agrego en el constructor de **Automovil**?

# Constructores

El compilador Java cuando compila una clase, **agrega el constructor nulo**, si la clase no define un constructor e **inserta en todos los constructores una línea de código para invocar al constructor nulo de la superclase** si es que explícitamente no se invoca a un constructor específico de la superclase.

```
public class Vertebrado {  
    private int cantPatas;  
    public Vertebrado(){  
        System.out.println("Constructor de Vertebrado");  
    }  
    public void comer(){ }  
}
```

El compilador agrega `super();`

```
public class Mamifero extends Vertebrado {  
    public Mamifero(){  
        System.out.println("Constructor de Mamifero");  
    }  
    public void comer(){ }  
}
```

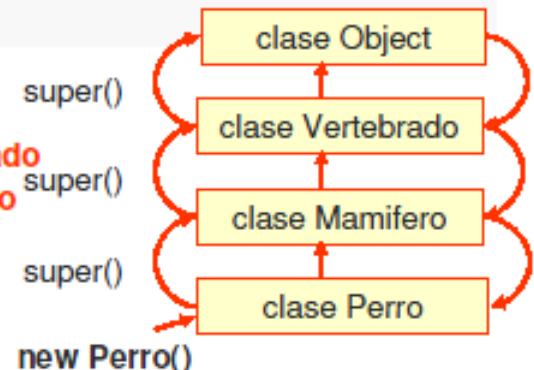
El compilador agrega `super();`

```
public class Perro extends Mamifero{  
    ..  
    public Perro(){  
        System.out.println("Constructor de Perro");  
    }  
    public void comer(){ }  
}
```

El compilador agrega `super();`

¿Cómo es la salida?

Constructor de Vertebrado  
Constructor de Mamifero  
Constructor de Perro





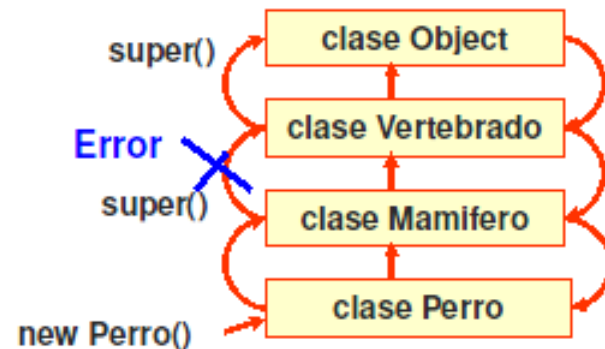
# Constructores

¿Qué pasa si Vertebrado declara solamente un constructor con argumentos?

```
public class Vertebrado {  
    private int cantPatas;  
    public Vertebrado(int c){  
        cantpatas= c;  
        System.out.println("Constructor de Mamifero");  
    }  
    public void comer(){ }  
}
```

```
public class Mamifero extends Vertebrado {  
    public Mamifero(){  
        System.out.println("Constructor de Mamifero");  
    }  
    public void comer(){ }  
}
```

```
public class Perro extends Mamifero{  
    public Perro(){  
        System.out.println("Constructor de Perro");  
    }  
    public void comer(){ }  
}
```



## ERROR DE COMPILACIÓN!!!

Desde el constructor de Mamifero NO se puede invocar al constructor nulo de Vertebrado porque no está definido

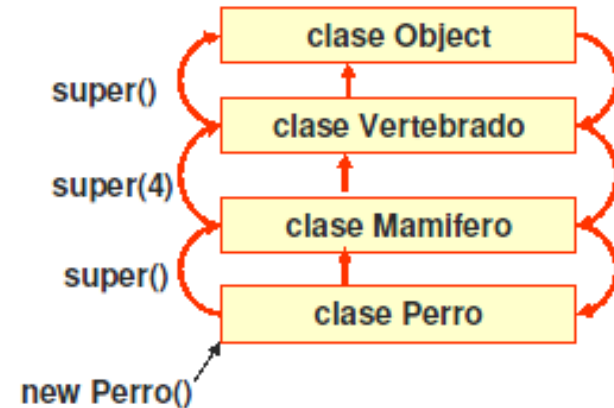
En este caso, es obligatorio invocar a un constructor definido **Vertebrado** usando la palabra clave **super(...)** y la lista de argumentos apropiada.

# Constructores

```
public class Vertebrado {  
    private int cantPatas;  
    public Vertebrado(int c){  
        .....>super();  
        cantpatas= c;  
        System.out.println("Constructor de Mamifero");  
    }  
    public void comer(){ }  
}
```

```
public class Mamifero extends Vertebrado {  
    public Mamifero(){  
        super(4);  
        System.out.println("Constructor de Mamifero");  
    }  
    public void comer(){ }  
}
```

```
public class Perro extends Mamifero{  
    public Perro(){  
        .....>super();  
        System.out.println("Constructor de Perro");  
    }  
    public void comer(){ }  
}
```



En cada constructor, el compilador inserta una línea de código para invocar al constructor nulo de la superclase: `super()`.

En caso de no estar definido el constructor nulo en la superclase, es obligatorio invocar explícitamente a un constructor con argumentos de la superclase en la primera línea de código del constructor de la subclase.

# Constructores

Si la clase Vehiculo contiene un único constructor con 2 argumentos strings, el compilador solamente permitirá crear objetos Vehiculo usando dicho constructor, en cualquier otro caso, el programa no compilará:

```
public class TestVehiculo {  
    public static void main(String[] args){  
        Vehiculo auto = new Vehiculo("EXQ056","Juan Perez");  
    }  
}
```

¿Qué pasa si se quieren construir objetos Vehiculo de distintas maneras?

Se declaran en la clase múltiples constructores: **sobrecarga de constructores**.

# Constructores. Sobrecarga

```
public class Vehiculo {  
    private String nroPatente="";  
    private String propietario="SinDueño";  
    public Vehiculo(){  
    }  
    public Vehiculo(String nroPatente){  
        this.nroPatente = nroPatente;  
    }  
    public Vehiculo(String nroPatente,String propietario){  
        this.nroPatente = nroPatente;  
        this.propietario = propietario;  
    }  
}
```

-La sobrecarga de constructores permite declarar múltiples versiones del constructor de la clase y de esta manera podemos crear e inicializar objetos de diferentes maneras.  
-El compilador determina el constructor a invocar a partir de la lista de argumentos.

```
public class TestVehiculo {  
    public static void main(String[] args){  
        Vehiculo a1=new Vehiculo();  
        Vehiculo a2=new Vehiculo("EXQ056");  
        Vehiculo a3=new Vehiculo("SJF034","Juan Ferrer");  
    }  
}
```

# Constructores. La palabra super

## super() y super(.....)

- Permite invocar a un constructor de la superclase.
- La invocación al constructor de la superclase debe hacerse en la primera línea de código del constructor de la clase derivada para garantizar que todos los datos de la superclase se inicialicen correctamente.

El código del constructor de **Perro** se terminará de ejecutar cuando se haya terminado de ejecutar el constructor de **Mamifero**.

```
public class Perro extends Mamifero {  
    private String sonido;  
    public Perro(){  
        super(4);  
        sonido=new String("guau");  
    }  
    ...  
}
```

Se invoca al constructor de Mamifero con un argumento entero

# La palabra clave this

**this** es una referencia al objeto actual. Está disponible automáticamente en todos los métodos.

Sirve para eliminar la ambigüedad. Nos permite especificar a que apellido estoy haciendo referencia.

```
public class Persona {  
    private String apellido;  
    . . .  
  
    public void setApellido(String apellido){  
        this.apellido = apellido;  
    }  
  
    public static void main(String[] args){  
        Persona maria = new Persona();  
        maria.setApellido("Juárez");  
    }  
}
```

# La palabra clave this

Es muy común usar la palabra clave **this** dentro de los métodos de instancia de una clase, para referirse al objeto que está ejecutando el código.

¿Por qué queríamos usar **this**?

Típicamente dentro del cuerpo de un método nos podemos referir directamente a las variables miembros de un objeto por su nombre, sin embargo, a veces una variable miembro está oculta por un parámetro de un método que tiene el mismo nombre.


si no usamos el **this**, no asignará al atributo de la clase Persona el valor que recibe como parámetro.





```
package ejemplo;



public class Persona {
    private String apellido;
    . . .
    public void setApellido(String apellido){
        this.apellido = apellido;
    }
    . . . resto de los setters y getters
}
```





# Ejemplo

```
package ejemplo;  Declaración del paquete

public class Persona {  Comentario de una línea
    // variables de instancia
    private String apellido = "";  Declaración e
    private String nombre = "";  inicialización de
    private String ocupacion = "";  variables
    private int edad;

    /* misDatos devuelve un String con todos los datos
    de la persona*/  Comentario de varias líneas
    public String misDatos(){
        String datos = "";
        datos = "Apellido: "+this.getApellido()+
            " Nombre: "+this.getNombre()+
            " Ocupación: "+this.getOcupacion()+
            " Edad: " + this.getEdad();
        return datos;  Retorno del método
    }


    public String getApellido(){  Método getter
        return apellido;
    }

    public void setApellido(String apellido){
        this.apellido = apellido;  Método setter
    }
    ... resto de los setters y getters
}
```

```
package ejemplo;

public class TestPerona {

    public static void main(String[] args){
        Persona p = new Persona();
        p.setApellido(args[0]);
        p.setNombre(args[1]);
        p.setEdad(Integer.parseInt(args[2]));
        p.setOcupacion(args[3]);
        System.out.println(p.misDatos());
    }
}
```



Muestro en consola el resultado

Para ejecutar:

```
java TestPersona Jeréz María 23 Estudiante
```



# Clase Object.

## Metódo equals()

- ▶ Todo objeto en JAVA dispone de un método público llamado **equals()**, definido en la clase Object. Todas las clases lo heredan directa o indirectamente.

```
public boolean equals(Object obj){  
    return( true\false );  
}
```

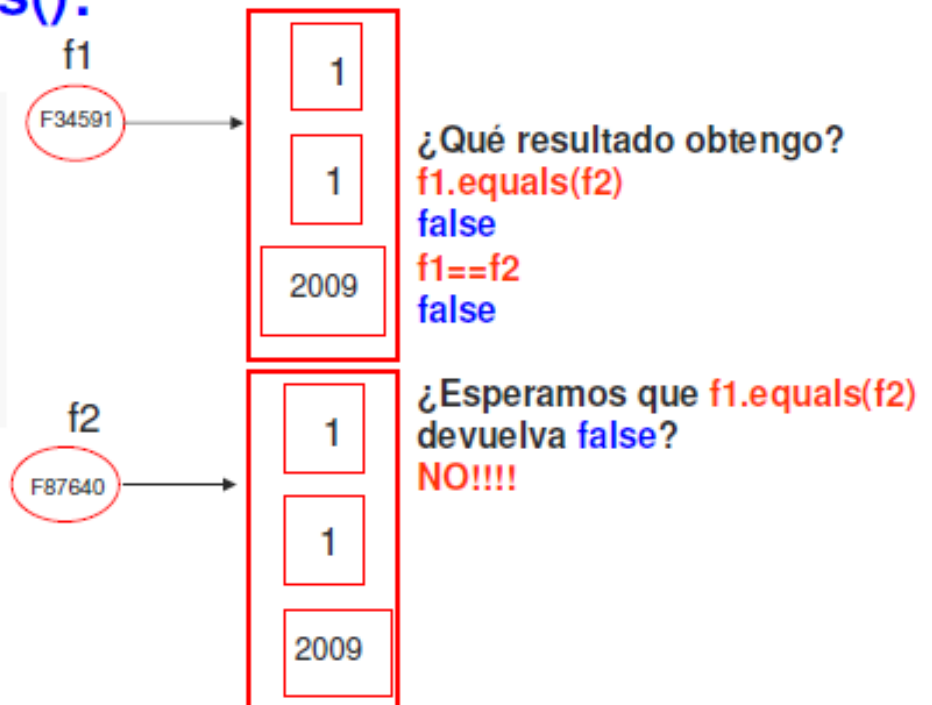
- ▶ La versión original compara referencias: devuelve **true** si la referencia del objeto recibido como argumento es igual a la del objeto receptor del mensaje, es decir si apuntan a la misma posición de memoria. NO compara el contenido de los objetos. Es equivalente a usar el operador **==**.
- ▶ Todos los objetos pueden responder a la invocación del método **equals()**. Es importante que cada una de las clases lo **sobreescriba** con el comportamiento deseado.

# Metódo equals()

## Ejemplo del método equals():

```
public class Fecha extends Object {  
    private int dia= 1;  
    private int mes= 1;  
    private int año=2009;  
    // métodos de instancia  
}
```

Creo dos fechas:  
Fecha f1 = new Fecha();  
Fecha f2 = new Fecha();



# Metódo equals()

Sobreescribimos el método equals():

```
public class Fecha extends Object {  
    private int dia = 1;  
    private int mes = 1;  
    private int año = 2009;
```

Es un operador que permite  
determinar la clase real de un objeto

```
    public boolean equals(Object o){  
        boolean result=false;  
        if ((o!=null) && (o instanceof Fecha)){  
            Fecha f=(Fecha) o;  
            if ((f.dia==this.dia)&&(f.mes==this.mes)&& (f.año==this.año)) result=true;  
        }  
        return result;  
    }  
}
```

Se castea el argumento o de  
Object a Fecha

COMPARA  
CONTENIDO

```
public static void main(String args[]){  
    Fecha f1, f2;  
    f1 = new Fecha();  
    f2 = new Fecha();  
    System.out.println(f1==f2);  
    System.out.println(f1.equals(f2));  
}
```

La salida es:  
**false**  
**true**

# Método hashCode()

El método hashCode devuelve un número.

Por lo tanto si dos objetos son iguales (según #equals) el valor retornado por sus respectivos hashCode debe ser igual.

Básicamente esta expresión debe ser siempre verdadera:  
`(!o1.equals(o2)) || (o1.hashCode() == o2.hashCode())`

Esto implica que si dos objetos no son iguales (según #equals) pueden tener o no el mismo valor de hashCode.

Implementar mal el hashCode puede pasar inadvertido pero es fundamental si se usan esos objetos dentro de las colecciones.

Este tema lo veremos en profundidad al ver Colecciones.

# Método toString()

- ▶ Otro método que todas las clases heredan de Object es el método **toString()**.
- ▶ El objetivo de este método es producir una representación textual y legible del contenido del objeto.

```
public String toString(){
```

- ▶ La versión original produce un string formado por el nombre de la clase, seguido del símbolo @ y de un valor hexadecimal calculado usando el objeto.

```
Fecha@25edc9
```

- ▶ La excepción es el literal **null** en cuyo caso el string producido es: **null**.
- ▶ De la misma manera que el método equals(), es importante que cada una de las clases lo **sobreescriba** con el comportamiento deseado.

# Método toString()

```
public class Fecha extends Object {  
    private int dia= 1;  
    private int mes= 1;  
    private int año=2009;  
    // métodos de instancia  
}
```

¿Cuál es la salida?

**Fecha@19821f**

No es información muy útil!!

Si tenemos un fragmento de código como el siguiente:

```
Fecha f1 = new Fecha();  
System.out.println(f1.toString());
```

No es necesario invocar al método `toString()` ya que los métodos `println()` y `print()` convierten a strings sus argumentos invocando al método `toString()`. En el caso que los argumentos sean datos primitivos se convierten a string usando el método `String.valueOf(int)` / `String.valueOf(long)` / `String.valueOf(boolean)`, etc.

Es equivalente a escribir **`System.out.println(f1);`**

# Método toString()

```
public class Persona extends Object{
    private String nombre = "Federico";
    private int edad = 25;
    public String toString(){
        return "Nombre: "+this.getNombre()+ " y edad: +this.getEdad();
    }
    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;
    }
}
```

Sobreescribimos el método toString() para obtener una salida particular de los objetos Persona

¿Cuál es la salida?

Federico

25

Nombre: Federico y edad: 25

```
public class TestPersona{
    public static void main (String args[]){
        Persona p = new Persona();
        System.out.println(p.getNombre());
        System.out.println(p.getEdad());
        System.out.println(p);
    }
}
```

# Operadores

Los datos se manipulan muchas veces utilizando operaciones con ellos. Los datos se suman, se restan, ... y a veces se realizan operaciones más complejas.



# Operadores aritméticos

operador	significado
+	Suma
-	Resta
*	Producto
/	División
%	Módulo (resto)

Hay que tener en cuenta que el resultado de estos operadores varía usamos enteros o si usamos números de coma flotante.

# Operadores aritméticos

Por ejemplo:

```
double resultado1, d1=14, d2=5;  
int resultado2, i1=14, i2=5;  
  
resultado1= d1 / d2;  
resultado2= i1 / i2;
```

*resultado1* valdrá 2.8 mientras que *resultado2* valdrá 2. Es más incluso:

```
double resultado;  
int i1=7,i2=2;  
resultado=i1/i2; //Resultado valdrá 3  
resultado=(double)i1/(double)i2; //Resultado valdrá 3.5
```

El operador del módulo (%) para calcular el resto de una división entera. Ejemplo:

```
int resultado, i1=14, i2=5;  
  
resultado = i1 % i2; //El resultado será 4
```

# Operadores condicionales

Sirven para comparar valores. Siempre devuelven valores booleanos. Son:

operador	significado
<	Menor
>	Mayor
>=	Mayor o igual
<=	Menor o igual
==	Igual
!=	Distinto
!	No lógico (NOT)
&&	"Y" lógico (AND)
	"O" lógico (OR)

# Operadores condicionales

Los operadores lógicos (AND, OR y NOT), sirven para evaluar condiciones complejas. NOT sirve para negar una condición. Ejemplo:

```
boolean mayorDeEdad, menorDeEdad;  
int edad = 21;  
mayorDeEdad = edad >= 18; //mayorDeEdad será true  
menorDeEdad = !mayorDeEdad; //menorDeEdad será false
```

El operador && (AND) sirve para evaluar dos expresiones de modo que si ambas son ciertas, el resultado será **true** sino el resultado será **false**. Ejemplo:

```
boolean carnetConducir=true;  
int edad=20;  
boolean puedeConducir= (edad>=18) && carnetConducir;  
//Si la edad es de al menos 18 años y carnetConducir es  
//true, puedeConducir es true
```

El operador || (OR) sirve también para evaluar dos expresiones. El resultado será **true** si al menos uno de las expresiones es **true**. Ejemplo:

```
boolean nieva =true, llueve=false, graniza=false;  
boolean malTiempo= nieva || llueve || graniza;
```

# Operadores de asignación

Permiten asignar valores a una variable. El fundamental es “=”. Pero sin embargo se pueden usar expresiones más complejas como:

```
x += 3;
```

En el ejemplo anterior lo que se hace es sumar 3 a la x (es lo mismo  $x+=3$ , que  $x=x+3$ ). Eso se puede hacer también con todos estos operadores:

$+=$

$-=$

$*=$

$/=$

$\&=$

$|=$

$\wedge=$

$\%=$

$>>=$

$<<=$

# Operadores de asignación

También se pueden concatenar asignaciones:

```
x1 = x2 = x3 = 5;
```

Otros operadores de asignación son “++” (incremento) y “--” (decremento). Ejemplo:

```
x++; //esto es x=x+1;  
x--; //esto es x=x-1;
```

Pero hay dos formas de utilizar el incremento y el decremento. Se puede usar por ejemplo `x++` o `++x`

La diferencia estriba en el modo en el que se comporta la asignación. Ejemplo:

```
int x=5, y=5, z;  
z=x++; //z vale 5, x vale 6  
z=++y; //z vale 6, y vale 6
```

# Operador ?

- ▶ Este operador (conocido como **if de una línea**) permite **ejecutar una instrucción u otra** según el valor de la expresión. Sintaxis:
- ▶ `expresionlogica?valorSiVerdadero:valorSiFalso;`

# Operador ?

Ej: `paga=(edad>18)?6000:3000;`

En este caso si la variable edad es mayor de 18, la paga será de 6000, sino será de 3000.

Se evalúa una condición y según es cierta o no se devuelve un valor u otro. Nótese que esta función ha de devolver un valor y no una expresión correcta.

Es decir, no funcionaría:

► `(edad>18)? paga=6000: paga=3000;`  
***/ERROR!!!!***



# Precedencia

A veces hay expresiones con operadores que resultan confusas. Por ejemplo en:

► resultado =  $8 + 4 / 2$ ;

Es difícil saber el resultado. ¿Cuál es? ¿seis o diez? La respuesta es 10 y la razón es que el operador de división siempre precede en el orden de ejecución al de la suma. Es decir, siempre se ejecuta antes la división que la suma. Siempre se pueden usar paréntesis para forzar el orden deseado:

► resultado =  $(8 + 4) / 2$ ;

# Precedencia. Orden.

operador			
0	[]	.	
++	--	~	!
*	/	%	
+	-		
>>	>>>	<<	<<<
>	>=	<	<=
==	!=		
&			
^			
&&			
?:			
=	+=, -=, *=, ...		

En la tabla anterior los operadores con mayor precedencia está en la parte superior, los de menor precedencia en la parte inferior. De izquierda a derecha la precedencia es la misma. Es decir, tiene la misma precedencia el operador de suma que el de resta.

# Precedencia

Esto último provoca conflictos, por ejemplo en:

► resultado = 9 / 3 \* 3;

El resultado podría ser uno ó nueve. En este caso el resultado es nueve, porque la división y el producto tienen la misma precedencia; por ello el compilador de Java realiza primero la operación que este más a la izquierda, que en este caso es la división.

Una vez más los paréntesis podrían evitar estos conflictos.

# la clase Math

Se echan de menos operadores matemáticos más potentes en Java. Por ello se ha incluido una clase especial llamada **Math** dentro del paquete **java.lang**. Para poder utilizar esta clase, se debe incluir esta instrucción:

```
import java.lang.Math;
```

Esta clase posee métodos muy interesantes para realizar cálculos matemáticos complejos. Por ejemplo:

```
double x= Math.pow(3,3); //x es 33
```

**Math** posee dos constantes, que son:

constante	significado
<b>double E</b>	El número <b>e</b> (2, 7182818245...)
<b>double PI</b>	El número <b>Π</b> (3,14159265...)

# Estructuras de control del flujo

# if

Permite crear estructuras condicionales simples; en las que al cumplirse una condición se ejecutan una serie de instrucciones. Se puede hacer que otro conjunto de instrucciones se ejecute si la condición es falsa. La condición es cualquier expresión que devuelva un resultado de **true** o **false**. **La sintaxis de la instrucción if es:**

```
if (condición) {  
    instrucciones que se ejecutan si la condición es true  
}  
else {  
    instrucciones que se ejecutan si la condición es false  
}
```



# if

**Ejemplo:**

```
if ((diasemana>=1) && (diasemana<=5)){  
    trabajar = true;  
}  
else {  
    trabajar = false;  
}
```

# switch

Es la estructura condicional compleja porque permite evaluar varios valores a la vez.  
Sintaxis:

```
switch (expresión) {  
    case valor1:  
        sentencias si la expresion es igual al valor1;  
        [break]  
    case valor2:  
        sentencias si la expresion es igual al valor2;  
        [break]  
    .  
    .  
    .  
    default:  
        sentencias que se ejecutan si no se cumple ninguna  
        de las anteriores  
}
```



# switch

- ▶ Esta instrucción evalúa una expresión (que debe ser short, int, byte o char), y según el valor de la misma ejecuta instrucciones. Cada case contiene un valor de la expresión si efectivamente la expresión equivale a ese valor, se ejecutan las instrucciones de ese case y de los siguientes.
- ▶ La instrucción **break** se utiliza para salir del switch. De tal modo que si queremos que para un determinado valor se ejecuten las instrucciones de un apartado case y sólo las de ese apartado, entonces habrá que finalizar ese case con un break.
- ▶ El bloque **default** sirve para ejecutar instrucciones para los casos en los que la expresión no se ajuste a ningún case.

# while

- ▶ La instrucción **while** permite crear bucles. Un bucle es un conjunto de sentencias que se repiten si se cumple una determinada condición.
- ▶ Los bucles **while** agrupan instrucciones las cuales se ejecutan continuamente hasta que una condición que se evalúa sea falsa.
- ▶ La condición se mira antes de entrar dentro del while y cada vez que se termina de ejecutar las instrucciones del while

# while

Sintaxis:

```
while (condición) {  
    sentencias que se ejecutan si la condición es true  
}
```

Ejemplo (cálculo del factorial de un número, el factorial de 4 sería:  $4*3*2*1$ ):

```
//factorial de 4  
int n=4, factorial=1, temporal=n;  
  
while (temporal>0) {  
    factorial*=temporal--;  
}
```

# do while

Crea un bucle muy similar al anterior. La diferencia es que la condición se evalúa después de ejecutar las instrucciones, lo que hace que al menos el bucle se ejecuta una vez.

Sintaxis:

```
do {  
    instrucciones  
} while (condición)
```

# for

- ▶ Es un bucle más complejo especialmente pensado para rellenar arrays o para ejecutar instrucciones controladas por un contador.
- ▶ Una vez más se ejecutan una serie de instrucciones en el caso de que se cumpla una determinada condición. Sintaxis:

```
for (expresiónInicial; condición;  
      ExpresiónEncadavuelta) {  
    instrucciones;  
}
```

# for

- ▶ La **expresión inicial es una instrucción que se ejecuta una sola vez: al entrar por primera vez en el bucle for (normalmente esa expresión lo que hace es dar valor inicial al contador del bucle).**
- ▶ La **condición es cualquier expresión que devuelve un valor lógico. En el caso de que** esa expresión sea verdadera se ejecutan las instrucciones. Cuando la condición pasa a ser falsa, el bucle deja de ejecutarse. La condición se valora cada vez que se terminan de ejecutar las instrucciones del bucle.
- ▶ Después de ejecutarse las instrucciones interiores del bucle, se realiza la expresión que tiene lugar tras ejecutarse las instrucciones del bucle (que, generalmente,
- ▶ incrementa o decrementa al contador). Luego se vuelve a evaluar la condición y así sucesivamente hasta que la condición sea falsa.

# for

Ejemplo (factorial):

```
//factorial de 4
int n=4, factorial=1, temporal=n;

for (temporal=n;temporal>0;temporal--){
    factorial *=temporal;
}
```

# sentencias de salida de un bucle

## ► break

Es una sentencia que permite salir del bucle en el que se encuentra inmediatamente.

Hay que intentar evitar su uso ya que produce malos hábitos al programar.



# sentencias de salida de un bucle

## ► continue

Instrucción que siempre va colocada dentro de un bucle y que hace que el flujo del programa ignore el resto de instrucciones del bucle; dicho de otra forma, va hasta la siguiente iteración del bucle.

Al igual que ocurría con **break**, **hay que intentar evitar su uso.**

# arrays

- ▶ Un array es una colección de valores de un mismo tipo agrupados en la misma variable.
- ▶ De forma que se puede acceder a cada valor independientemente.
- ▶ Para Java además un array es un objeto que tiene propiedades que se pueden manipular.

# arrays

La declaración de un array unidimensional se hace con esta sintaxis.

```
tipo nombre[];
```

Ejemplo:

```
double cuentas[]; //Declara un array que almacenará valores  
                  // doubles
```

Declara un array de tipo double. Esta declaración indica para qué servirá el array, pero no reserva espacio en la RAM al no saberse todavía el tamaño del mismo.

# arrays

Tras la declaración del array, se tiene que iniciar. Eso lo realiza el operador **new**, que es el que realmente crea el array indicando un tamaño. Cuando se usa **new** es cuando se reserva el espacio necesario en memoria. Un array no inicializado es un array **null**. Ejemplo:

```
int notas[]; //sería válido también int[] notas;  
notas = new int[3]; //indica que el array constará de tres  
                  //valores de tipo int  
  
//También se puede hacer todo a la vez  
//int notas[]=new int[3];
```

En el ejemplo anterior se crea un array de tres enteros (con los tipos básicos se crea en memoria el array y se inicializan los valores, los números se inician a 0).

# arrays

Los valores del array se asignan utilizando el índice del mismo entre corchetes:

```
notas[2]=8;
```

También se pueden asignar valores al array en la propia declaración:

```
int notas[] = {8, 7, 9};  
int notas2[]= new int[] {8,7,9};//Equivalente a la anterior
```

Esto declara e inicializa un array de tres elementos. En el ejemplo lo que significa es que `notas[0]` vale 8, `notas[1]` vale 7 y `notas[2]` vale 9.

En Java (como en otros lenguajes) el primer elemento de un array es el cero. El primer elemento del array `notas`, es `notas[0]`. Se pueden declarar arrays a cualquier tipo de datos (enteros, booleanos, doubles, ... e incluso objetos).

# arrays

La ventaja de usar arrays (volviendo al caso de las notas) es que gracias a un bucle **for** se puede rellenar o leer fácilmente todos los elementos de un array:

```
//Calcular la media de las 18 notas
suma=0;
for (int i=0;i<=17;i++){
    suma+=nota[i];
}
media=suma/18;
```

# arrays

A un array se le puede inicializar las veces que haga falta:

```
int notas[]=new notas[16];  
...  
notas=new notas[25];
```

Pero hay que tener en cuenta que el segundo new hace que se pierda el contenido anterior. Realmente un array es una referencia a valores que se almacenan en memoria mediante el operador new, si el operador **new** se utiliza en la misma referencia, el anterior contenido se queda sin referencia y, por lo tanto se pierde.

# arrays

Un array se puede asignar a otro array (si son del mismo tipo):

```
int notas[];  
int ejemplo[]=new int[18];  
notas=ejemplo;
```

En el último punto, notas equivale a ejemplo. Esta asignación provoca que cualquier cambio en notas también cambie el array ejemplos. Es decir esta asignación anterior, no copia los valores del array, sino que notas y ejemplo son referencias al mismo array. Ejemplo:

```
int notas[]={3,3,3};  
int ejemplo[]=notas;  
ejemplo= notas;  
ejemplo[0]=8;  
System.out.println(notas[0]);//Escribirá el número 8
```



# arrays multidimensionales

Los arrays además pueden tener varias dimensiones. Entonces se habla de arrays de arrays (arrays que contienen arrays) Ejemplo:

```
int notas[] [];
```

*notas* es un array que contiene arrays de enteros

```
notas = new int[3][12]; //notas está compuesto por 3 arrays  
                        //de 12 enteros cada uno  
notas[0][0]=9; //el primer valor es 0
```

# arrays multidimensionales

Puede haber más dimensiones incluso (notas[3][2][7]). Los arrays multidimensionales se pueden inicializar de forma más creativa incluso. Ejemplo:

```
int notas[][]=new int[5][]; //Hay 5 arrays de enteros
notas[0]=new int[100]; //El primer array es de 100 enteros
notas[1]=new int[230]; //El segundo de 230
notas[2]=new int[400];
notas[3]=new int[100];
notas[4]=new int[200];
```

Hay que tener en cuenta que en el ejemplo anterior, notas[0] es un array de 100 enteros. Mientras que notas, es un array de 5 arrays de enteros.

Se pueden utilizar más de dos dimensiones si es necesario.

# longitud de un array

- ▶ Los arrays poseen un método que permite determinar cuánto mide un array.
- ▶ Se trata de **length**. **Ejemplo (continuando del anterior):**
- ▶ **`System.out.println(notas.length); //Sale 5`**
- ▶ **`System.out.println(notas[2].length); //Sale 400`**

# la clase Arrays

- ▶ En el paquete **java.util** se encuentra una clase **estática llamada Arrays**. Una clase estática permite ser utilizada como si fuera un objeto (como ocurre con **Math**).
- ▶ **Esta** clase posee métodos muy interesantes para utilizar sobre arrays.
- ▶ Su uso es **Arrays.método(argumentos)**;

# la clase Arrays

## fill

Permite rellenar todo un array unidimensional con un determinado valor. Sus argumentos son el array a rellenar y el valor deseado:

```
int valores[]=new int[23];  
Arrays.fill(valores,-1);//Todo el array vale -1
```

También permite decidir desde que índice hasta qué índice rellenamos:

```
Arrays.fill(valores,5,8,-1);//Del elemento 5 al 7 valdrán -1
```

## equals

Compara dos arrays y devuelve true si son iguales. Se consideran iguales si son del mismo tipo, tamaño y contienen los mismos valores.

# la clase Arrays

## sort

Permite ordenar un array en orden ascendente. Se pueden ordenar sólo una serie de elementos desde un determinado punto hasta un determinado punto.

```
int x[]={4,5,2,3,7,8,2,3,9,5};  
Arrays.sort(x);//Estará ordenado  
Arrays.sort(x,2,5);//Ordena del 2º al 4º elemento
```

## binarySearch

Permite buscar un elemento de forma ultrarrápida en un array ordenado (en un array desordenado sus resultados son impredecibles). Devuelve el índice en el que está colocado el elemento. Ejemplo:

```
int x[]={1,2,3,4,5,6,7,8,9,10,11,12};  
Arrays.sort(x);  
System.out.println(Arrays.binarySearch(x,8));//Da 7
```

# el método System.arrayCopy

La clase System también posee un método relacionado con los arrays, dicho método permite copiar un array en otro. Recibe cinco argumentos: el array que se copia, el índice desde que se empieza a copia en el origen, el array destino de la copia, el índice desde el que se copia en el destino, y el tamaño de la copia (número de elementos de la copia).

```
int uno[]={1,1,2};
int dos[]={3,3,3,3,3,3,3,3,3};
System.arraycopy(uno, 0, dos, 0, uno.length);
for (int i=0;i<=8;i++){
    System.out.print(dos[i]+" ");
} //Sale 112333333
```

# For each

Esta estructura nos permite recorrer una Colección o un array de elementos de una forma sencilla. Evitando el uso de un bucle for normal.

De la forma tradicional podríamos recorrer un array de la siguiente

```
String a[] = {"Avila", "Burgos", "León", "Palencia",  
"Salamanca", "Segovia", "Soria", "Valladolid",  
"Zamora"};
```

```
for (int x=0; x<a.length; x++)  
    System.out.println(a[x]);
```



# For each

En el caso anterior nos estamos apoyando en el tamaño del array, con la propiedad `length` y en una variable contador, la cual vamos incrementando hasta que llegue a el tamaño del array.

El bucle `for-each` en Javanos permite realizar estas mismas operaciones de una forma muy sencilla. La estructura del bucle `for-each` sería de la siguiente forma:

```
for (TipoBase variable: ArrayDeTiposBase)
{..}
```

## For each. Ejemplo

```
String array[] = {"Avila", "Burgos", "León",  
"Palencia", "Salamanca", "Segovia", "Soria",  
"Valladolid", "Zamora"};
```

```
for (String elemento: array)  
    System.out.println(elemento);
```