



# Sun Certified Java Programmer ( SCJP )

Noel Mamoghli





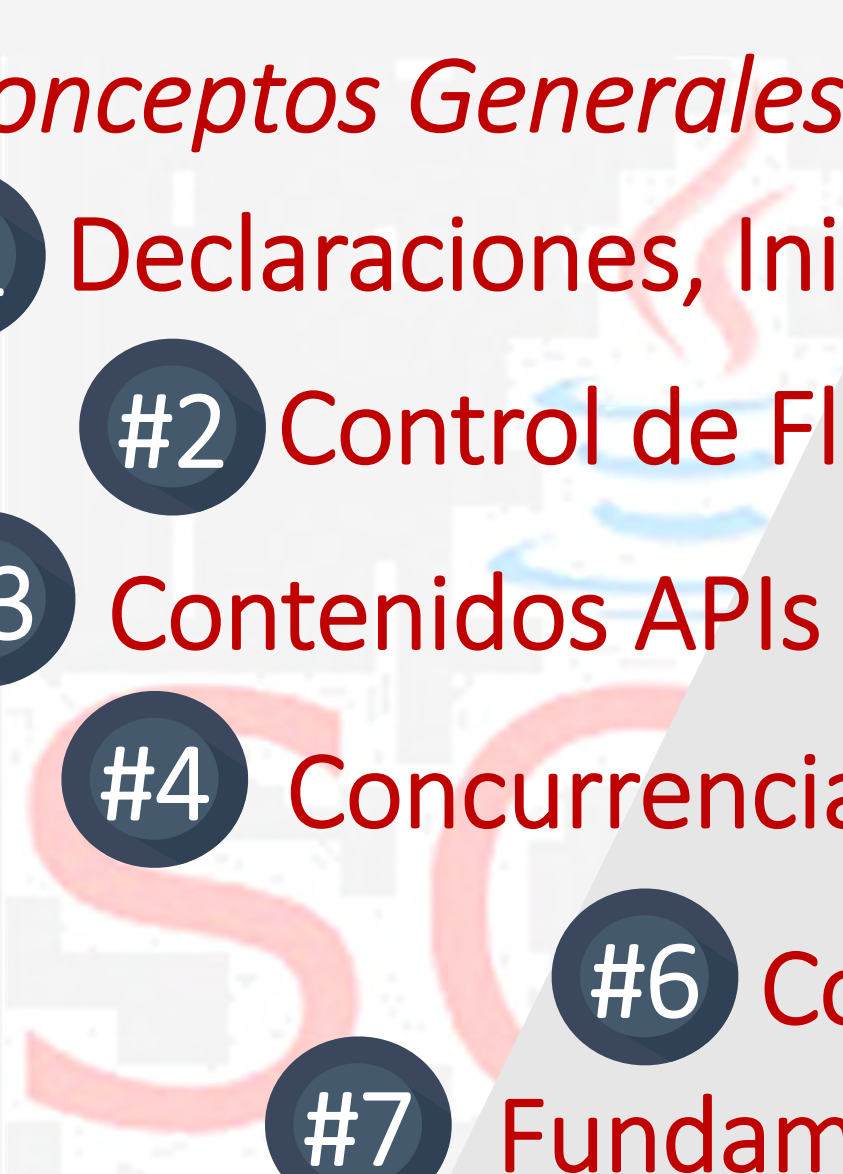
Noel Mamoghli

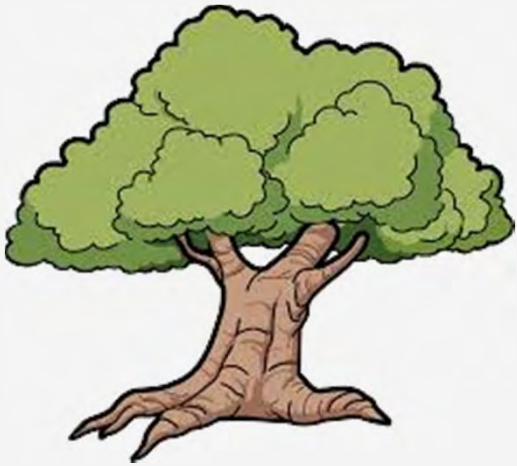


noel.nmd@gmail.com





- 
- #0 *Conceptos Generales e Infraestructura*
  - #1 Declaraciones, Inicialización y Ámbito
  - #2 Control de Flujo de Ejecución
  - #3 Contenidos APIs Principales
  - #4 Concurrencia
  - #5 Conceptos OO
  - #6 Colecciones y Genéricos
  - #7 Fundamentos



**James Gosling**



**THE NAME...**





**ACRONYM**

**James Gosling,  
Arthur  
Van Hoff, y  
Andy Bechtolsheim**



**ACRONYM**

**Just  
Another  
Vague  
Acronym**



**ACRONYM**



**Orientado a  
Objetos**



**Java™**

**Multi-  
Dispositivo**

**Maquina  
Virtual**

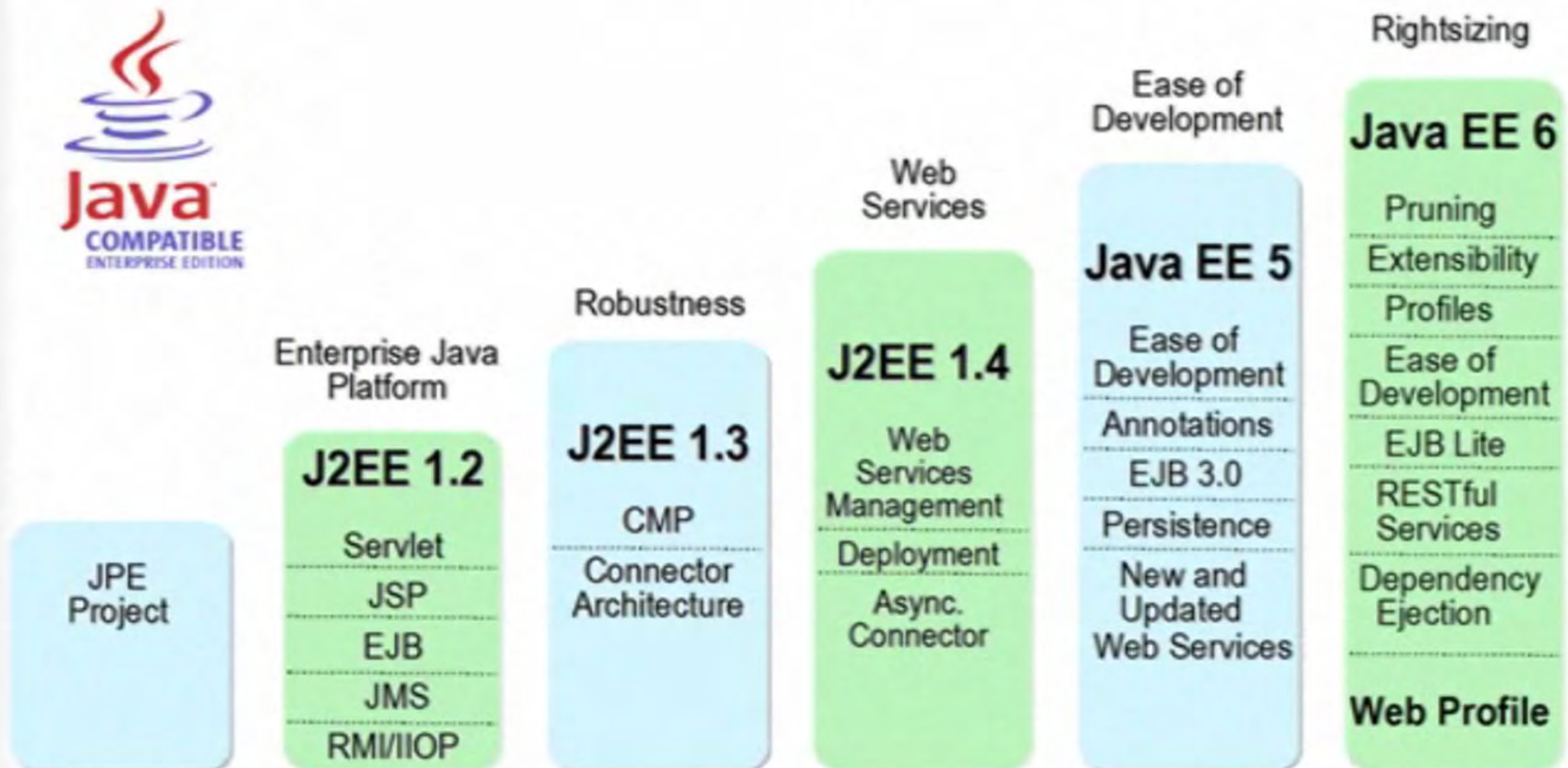
**Herencia  
Simple**

# Java SE Version History (Green: Major; Blue: Minor)





# Java EE: Past & Present



# Orientación a Objetos

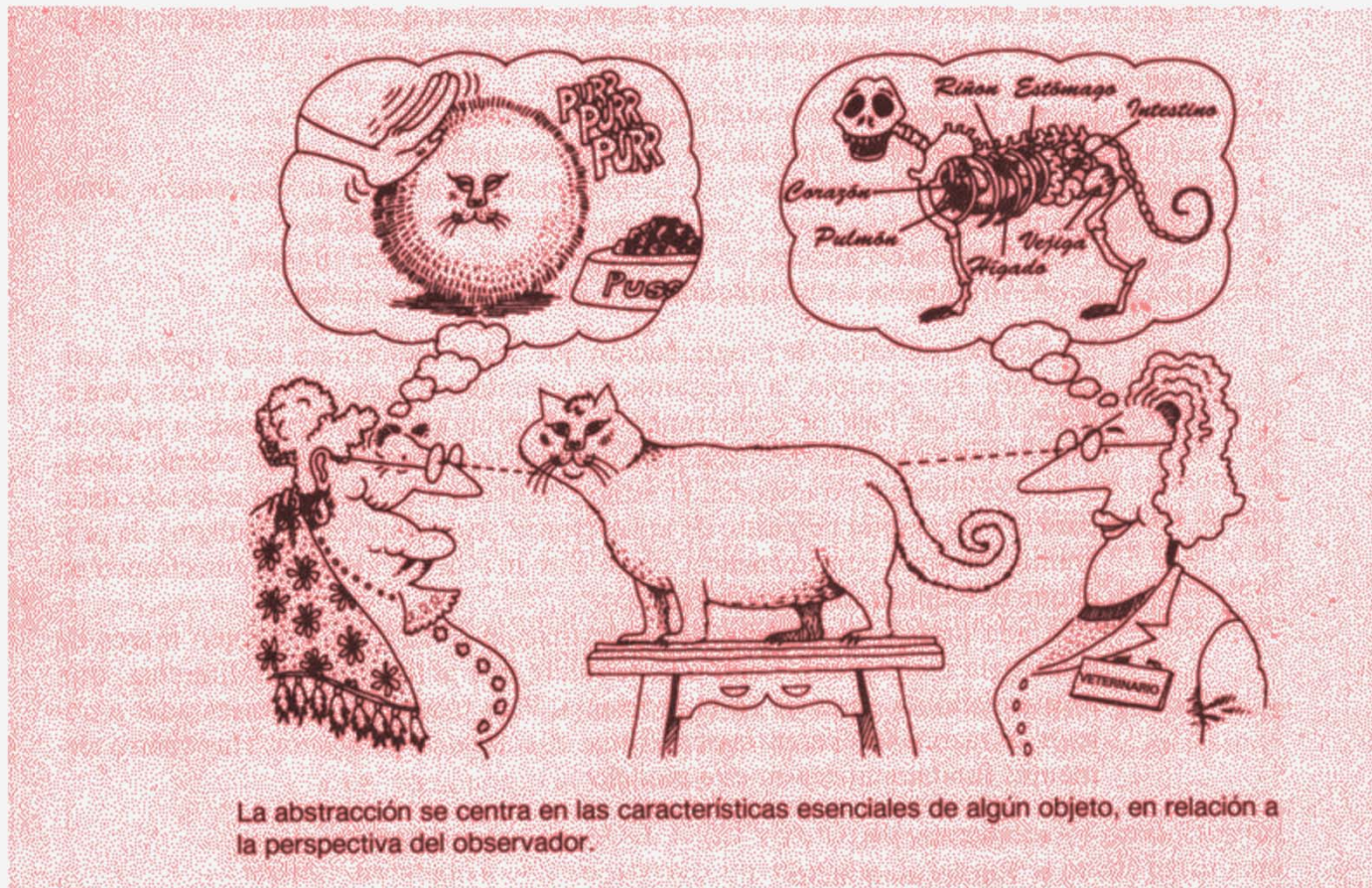


# Orientación a Objetos





# Orientación a Objetos

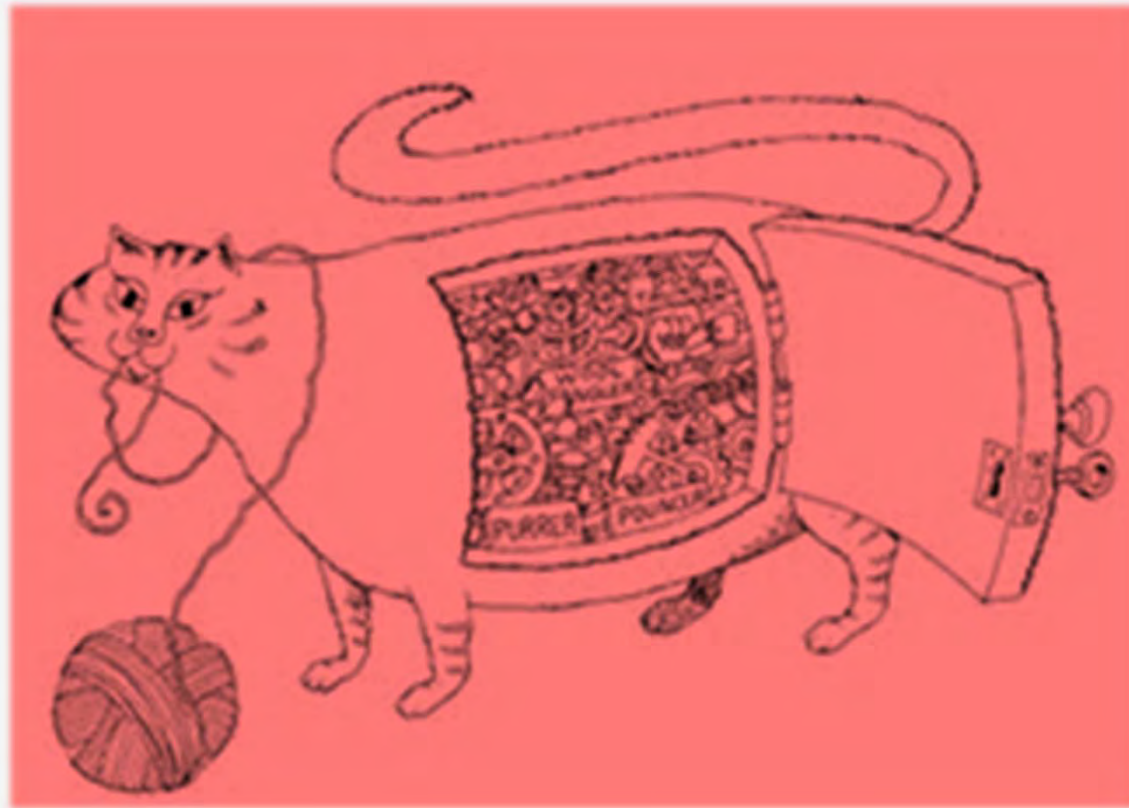


# Orientación a Objetos

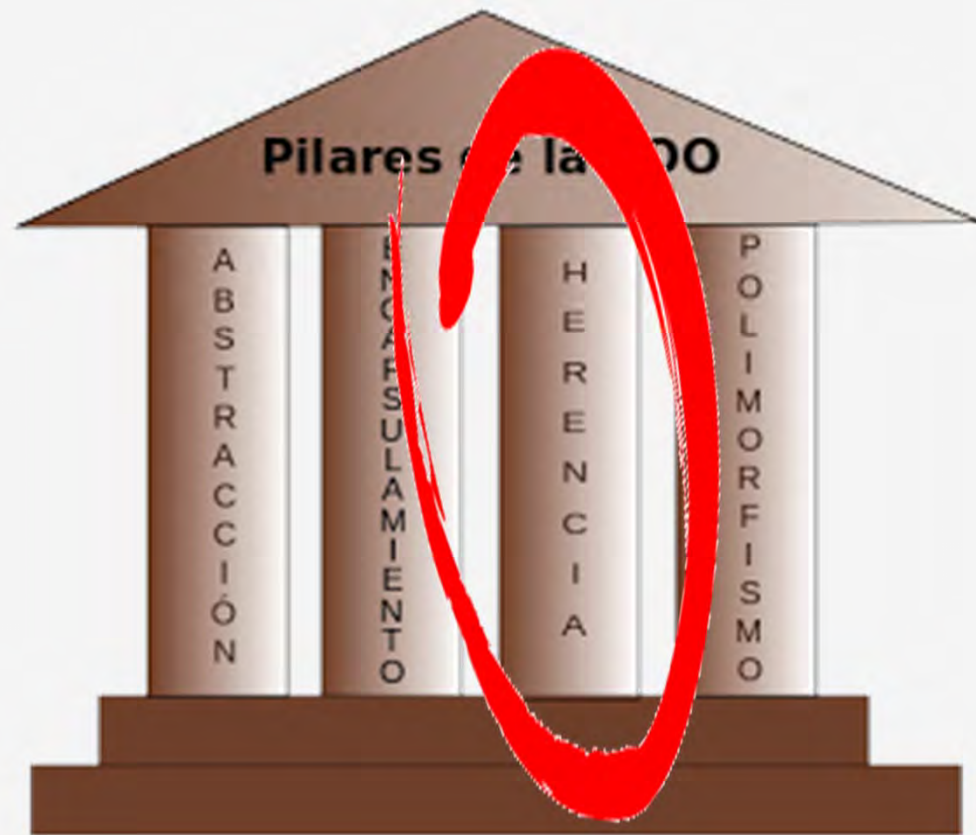




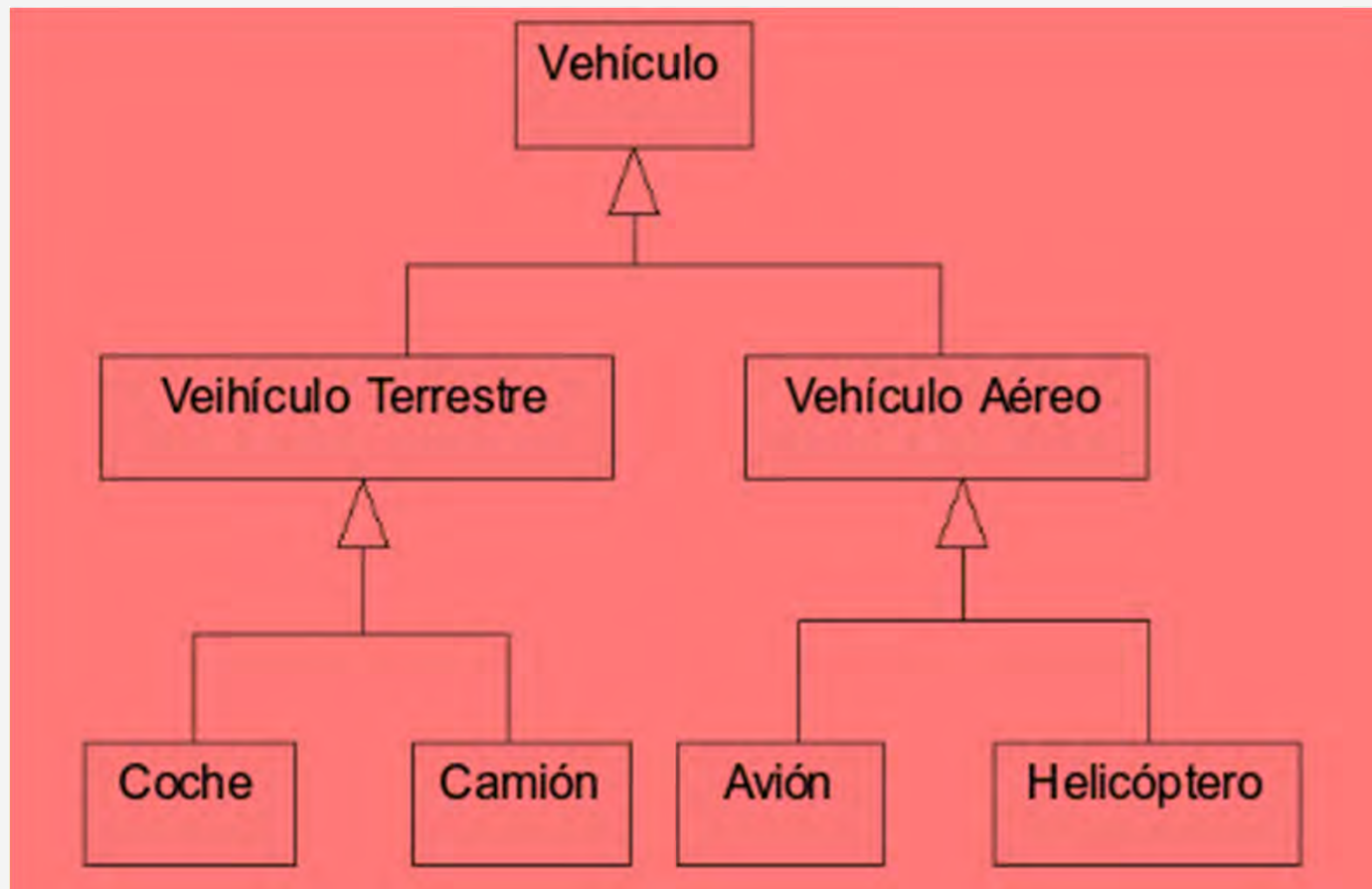
# Orientación a Objetos



# Orientación a Objetos



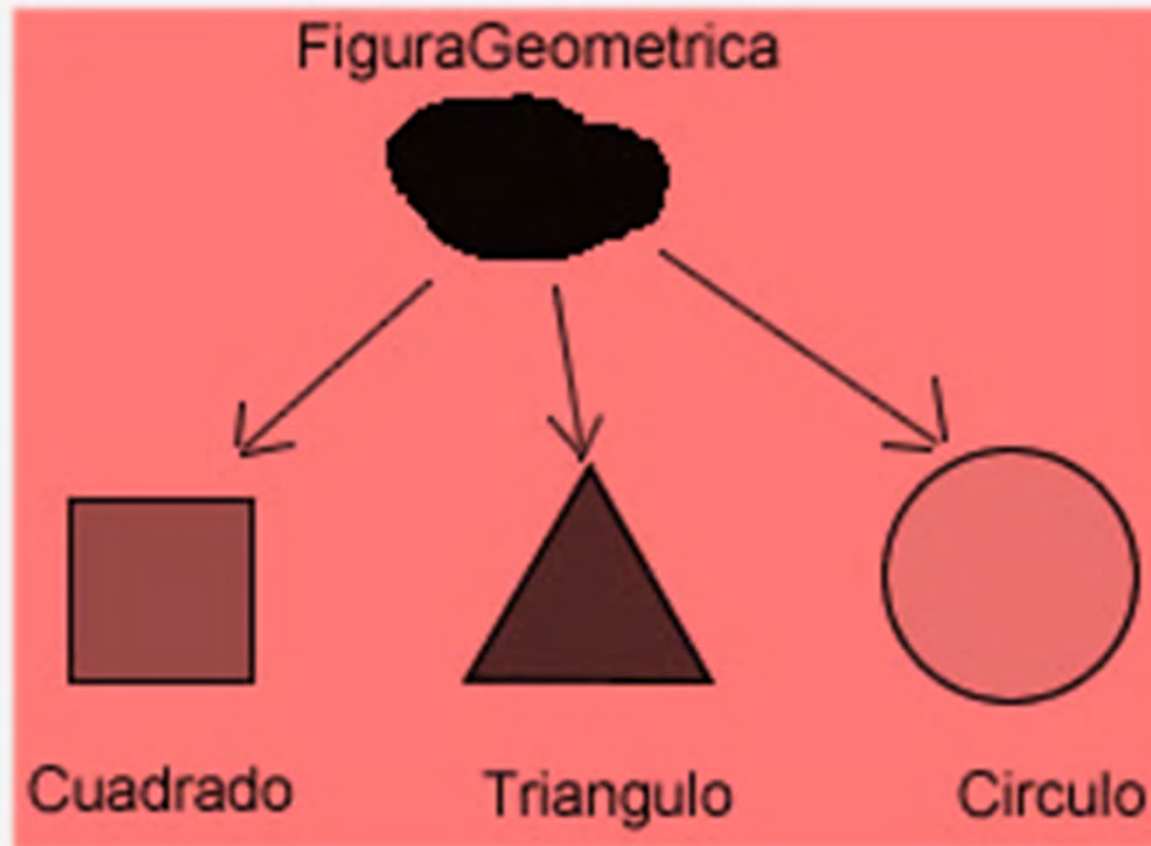
# Orientación a Objetos



# Orientación a Objetos

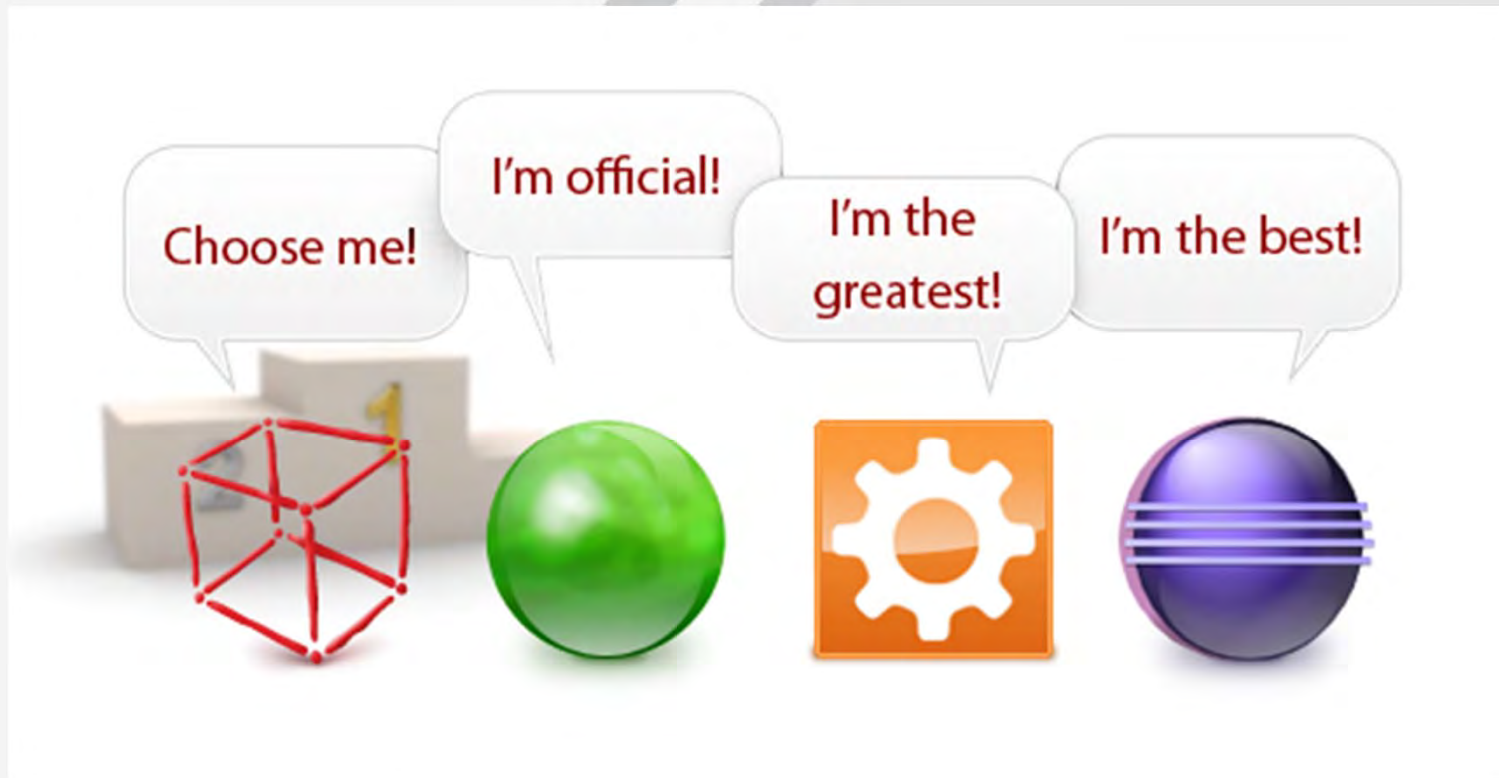


# Orientación a Objetos






# JAVA IDEs



# JAVA IDEs



Resaltado de sintaxis

Generación de código

Resaltado de errores y warning

Control de Versiones

Navegación

Soporte a otras tecnologías

Refactorización

Modo Debug

Complejidad de código

<https://netbeans.org/>



<https://eclipse.org/>



**hay otros...**

<http://www.jcreator.com/>



<https://www.jetbrains.com/idea/>



# Práctica: "JAVA IDEs"





# **Declaraciones, Inicialización y Ámbito**

# Class, Import y Package



- Sólo una clase pública por fuente
- Comentarios no cuentan
- Si hay clase pública, el nombre del fuente debe coincidir
- La sentencia *package* debe ser siempre la primera del fuente.

# Class, Import y Package



```
package es.hpe.scjp.section1;
```

```
import java.util.*;
```

```
import es.noelmd.something.*;
```

```
import this.is.anexample;
```

```
class FirstClass { ...}
```

```
class SecondClass { ...}
```

```
public class PublicClass {...}
```

# Acceso a clases

```
TemaOne.java x
package es.scjp.noelmd.c01;

public class TemaOne
{
    public static void main (String arfs [])
    {
        new ClassA();
    }
}

class ClassA
{
    private ClassB classB;
    private ClassC classC;

    public ClassA ()
    {
        System.out.println("Entrando en el constructor de ClassA");

        System.out.println("Creando instancia de ClassB");

        this.classB = new ClassB();

        System.out.println("Creando instancia de ClassC");

        this.classC = new ClassC();
    }
}

class ClassB
{
}

class ClassC extends ClassA
{
}
```



# Modificadores de Acceso



# Modificadores de Acceso default

Sin modificador específico.  
Cualquier clase del paquete puede  
acceder.



# Modificadores de Acceso

## public

Puede ser accedida desde cualquier clase que importe el paquete o la clase.



# Modificadores de Acceso

## final

La clase NO podrá ser extendida.



# Modificadores de Acceso

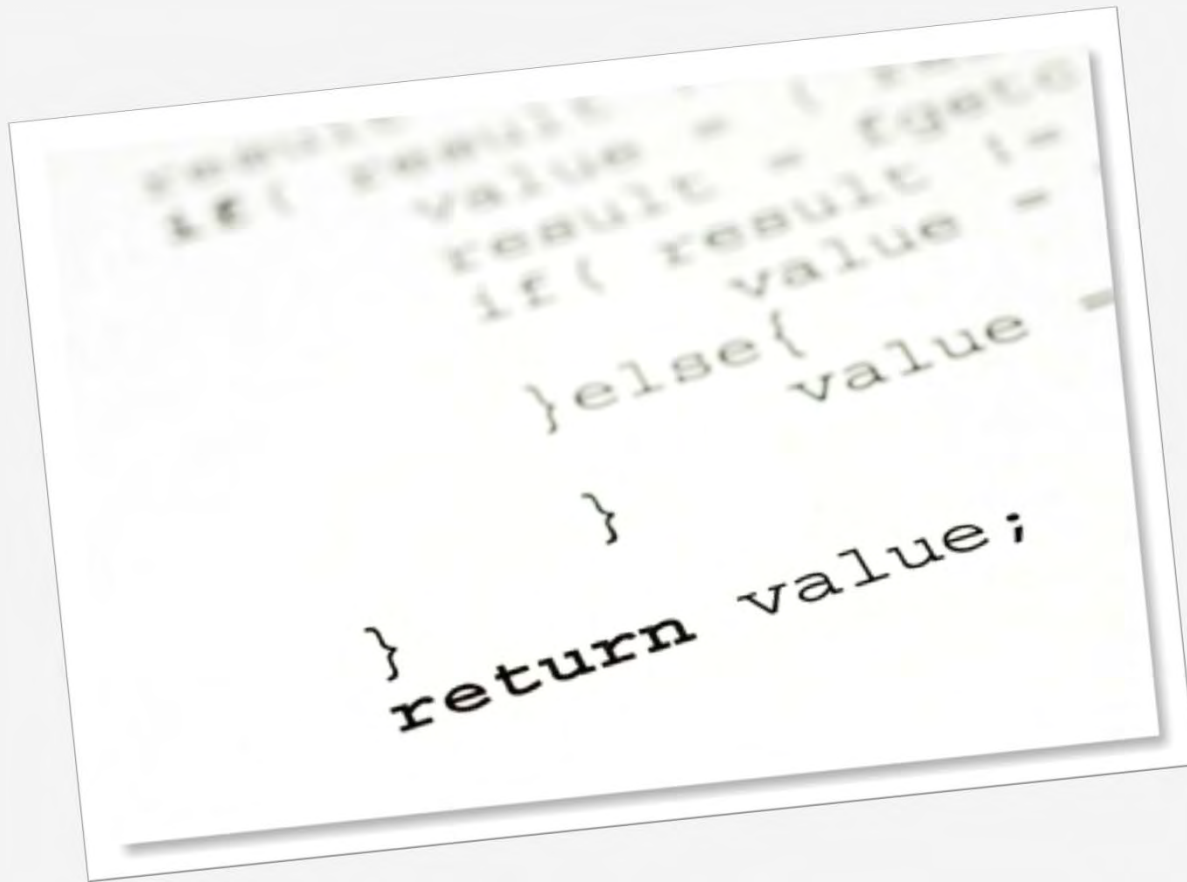
## abstract

La clase NO podrá  
ser instanciada.

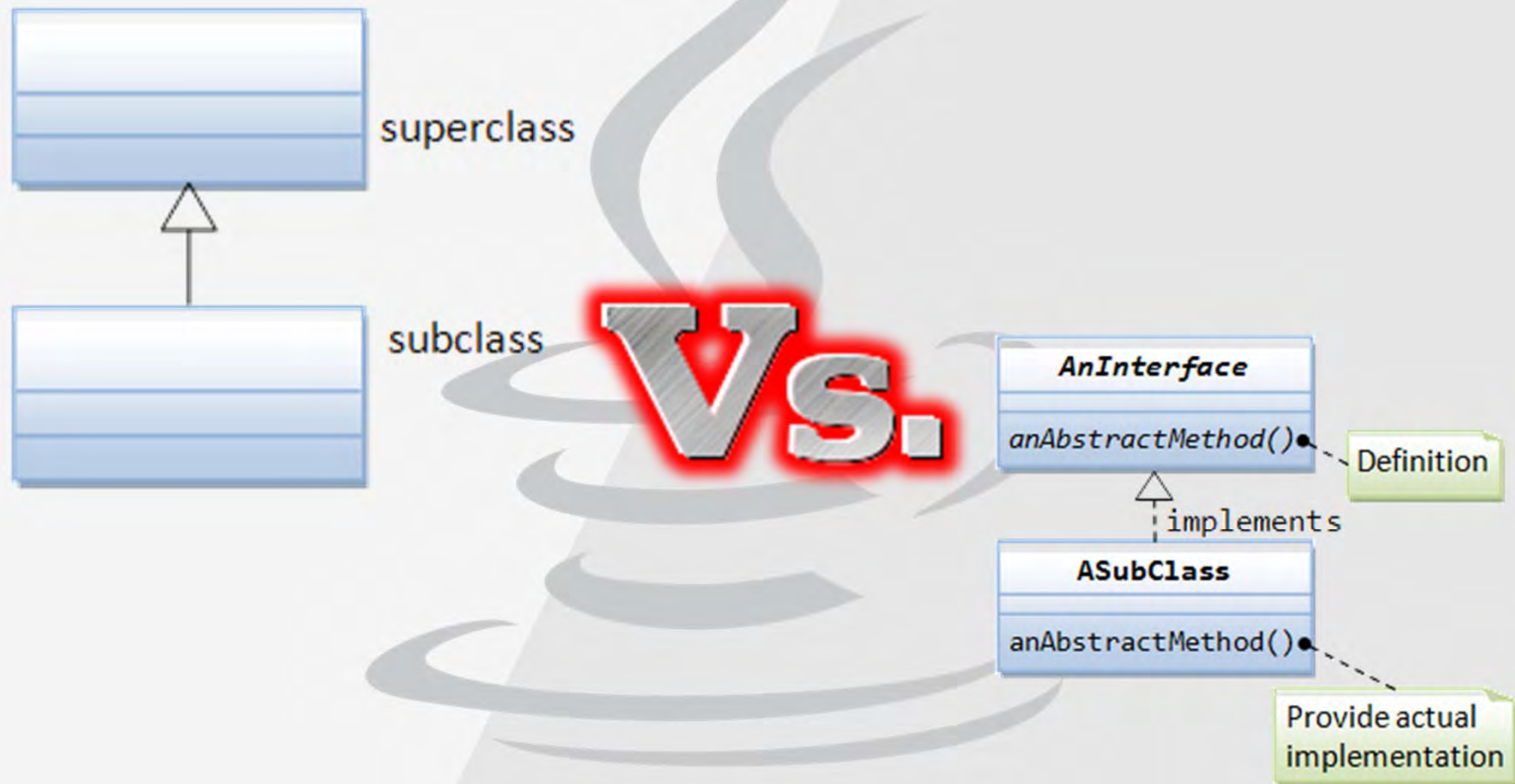




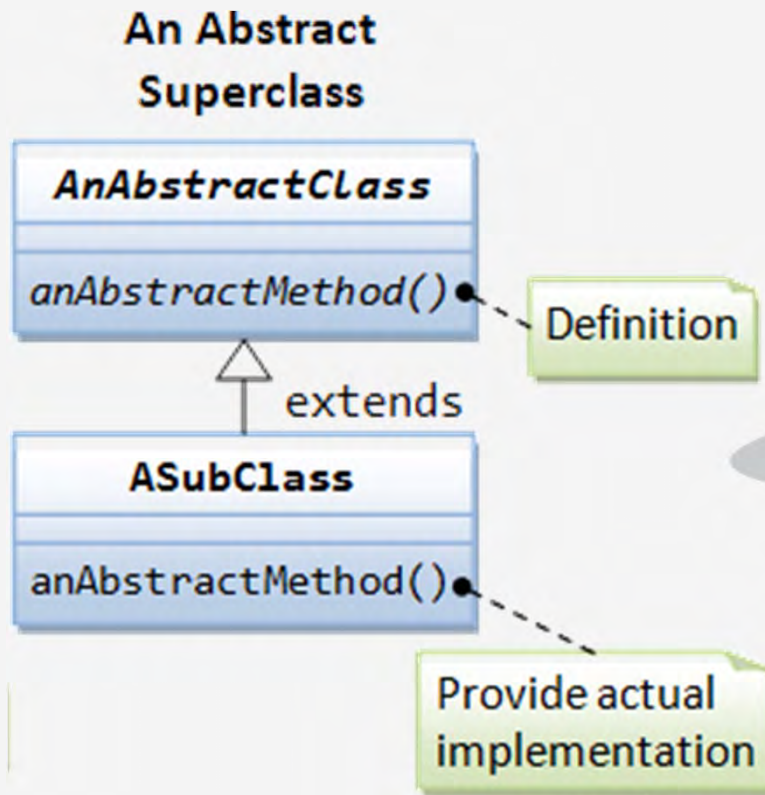
# Práctica: “ Modificadores Acceso a Clase ”



# Abstracción e Interfaces

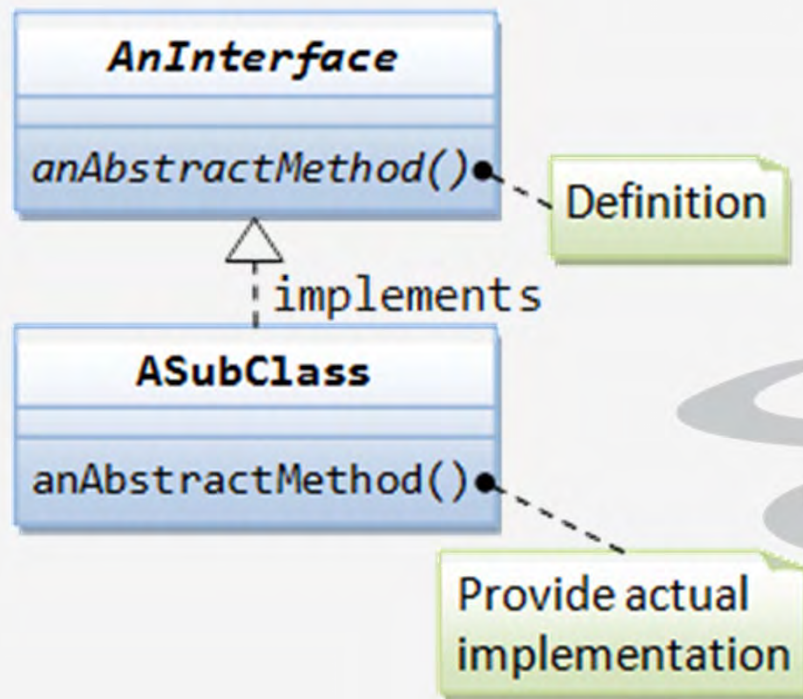


# Abstracción e Interfaces



- ➔ No es instanciable
- ➔ Métodos abstractos
- ➔ Métodos NO abstractos
- ➔ Subclases NO abstractas deben implementar métodos abstractos
- ➔ **Subclases abstractas**

# Abstracción e Interfaces



**Un interfaz es un contrato.**  
Especifica una serie de propiedades que debe cumplir quien acepta el contrato ( su implementación ).  
No dice nada en absoluto **CÓMO** debe cumplir el contrato.

**Una interfaz define el QUÉ, pero no el CÓMO.**

# Identificadores





# Identificadores

- ➔ Los identificadores de comenzar con una letra, el carácter \$ u otro carácter de unión como \_
- ➔ No pueden empezar con número.
- ➔ Después del primer carácter pueden incluir combinaciones de letras números y caracteres de unión.
- ➔ No hay límite en el número de caracteres.
- ➔ No se pueden utilizar palabras reservas del lenguaje
- ➔ **Son sensibles a mayúsculas – minúsculas (case-sensitive).**

# Modificadores de Acceso II



# Modificadores de Acceso II

## public

Si la clase que lo contiene  
es visible cualquiera puede  
acceder.



# Modificadores de Acceso II

```
package book;  
import cert.*;  
class Goo {  
    public static void main(String[] args) {  
        Sludge o = new Sludge();  
        o.testIt();  
    }  
}
```

```
package cert;  
public class Sludge {  
    public void testIt() {  
        System.out.println("sludge"); }  
}
```

# Modificadores de Acceso II

## private

No puede ser accedido por  
ningún mecanismo.

No se puede realizar  
“sobre-escritura”, solo es  
homónimo.





# Modificadores de Acceso II

```
package cert;
public class Roo {
    private String doRooThings() {
        // imagine the fun code that goes here, but
        // no other class will know
        return "fun";
    }
}

package cert;    //Cloo and Roo are in the same package
class Cloo extends Roo {
    //Still OK, superclass Roo is public
    public void testCloo() {
        System.out.println(doRooThings());
        //Compiler error!
    }
}
```

# Modificadores de Acceso II

## default

Sólo puede ser accedido  
por clases que se  
encuentran en el mismo  
paquete.



# Modificadores de Acceso II

## protected

Solo puede ser accedido por subclases de la clase que define este ámbito.



# Modificadores de Acceso II

Visibility	Public	Protected	Default	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, <i>through inheritance</i>	No	No
From any non-subclass class outside the package	Yes	No	No	No

# Modificadores de Acceso III





# Modificadores de Acceso III

final



# Modificadores de Acceso III

## final

Impide la sobrescritura de un método por las subclases de una superclase.



# Modificadores de Acceso III


```
package cert;
public class Suppa{
    public final String noMore() {
        // the code that goes here, but
        // no other subclass can override the method
        return "stop";
    }
}

package cert.fine.grained;
public class Child extends Suppa{
    @override
    public String noMore() {
        //Compiler error!
    }
}
```

# Modificadores de Acceso III

```
package cert;
public class Suppa{
    public final String noMore() {
        // the code that goes here, but
        // no other subclass can override the method
        return "stop";
    }
}

package cert.final.override;
public class ChupaCabra {
    @Override
    public String noMore() {
        //Comprobar si se puede seguir
    }
}
```



# Modificadores de Acceso III

## final

En un argumento obliga a que dicho valor se mantenga constante en dicho método.





# Modificadores de Acceso III

```
package cert;  
public class myClass {  
    public Record getRecord(int fileNumber,  
        final int recordNumber) {  
        recordNumber++;  
    }  
}
```

# Modificadores de Acceso III

```
package cert;  
public class myClass {  
    public Record getRecord(int fileNumber,  
        final Record recordNumber) {  
        recordNumber  
    }  
}
```



# Modificadores de Acceso III

## abstract

Método que ha sido  
declarado pero no  
implementado.  
Sólo propio de clases  
abstractas.



# Modificadores de Acceso III

```
package cert;  
public class IllegalClass{  
    public abstract void  
        doSomething();  
}
```

# Modificadores de Acceso III

```
package cert;  
public class IllegalClass{  
    public abstract void  
        doSomething();  
}
```





# Modificadores de Acceso III

## synchronized

Método que puede ser accedido por un único hilo concurrentemente.

Exclusivo para métodos  
(no para variables ni clases)



# Modificadores de Acceso III

```
package cert;  
  
public class EmployeeDTO  
{  
    public synchronized Record  
        updateUserInfo(UserInfo user) { }  
}
```

# Modificadores de Acceso III

## native

Método implementado en el lenguaje de la plataforma de la arquitectura.

Exclusivo para métodos (no variables ni clases)



# Modificadores de Acceso III

## transient

Indica a la JVM que una variable de clase no será tenida en cuenta cuando la clase vaya a ser serializada.



# Modificadores de Acceso III

## volatile

Una variable volatile implica que los diferentes hilos que la acceden sincronizan el último valor de dicha variable.





# Modificadores de Acceso III

## strictfp

Fuerza a utilizar operaciones de coma flotante conforme a la especificación IEEE 754.

Exclusivo de clase y método.



# Constructores



# Constructores



- Todas las clases deben tener al menos un constructor.
- Si no ponemos constructor se agrega el constructor por defecto.
- El constructor por defecto no tiene parámetros.
- Los constructores poseen el mismo nombre que la clase que los contiene.

# Constructores



- ➔ Todas las clases deben tener al menos un constructor.
- ➔ Si no ponemos constructor se agrega el constructor por defecto.
- ➔ El constructor por defecto no tiene parámetros.
- ➔ Los constructores poseen el mismo nombre que la clase que los contiene.
- ➔ Los constructores no devuelven nada.

# Constructores



- Pueden usar cualquier tipo de modificador.
- Pueden sobrecargarse.
- Las clases abstractas tienen constructores.
- Su primera sentencia implícita o explícita es siempre llamar a un constructor sobrecargado o al constructor super.
- Su ejecución es previa a cualquier llamada a método o acceso a variable de clase.
- Sólo puede invocarse un constructor desde otro constructor o instanciando la clase.



# Variables



# Variables

primitivo

char

boolean

byte

long

int

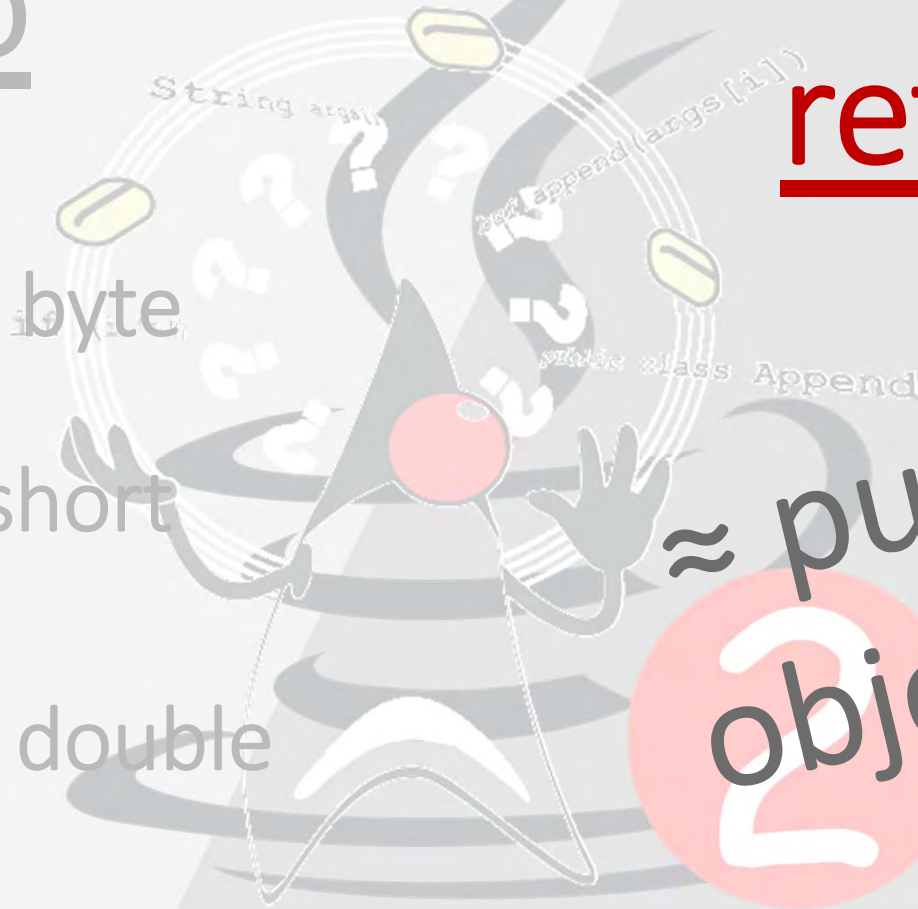
short

double

float

referencias

≈ punteros a  
objetos



# Variables

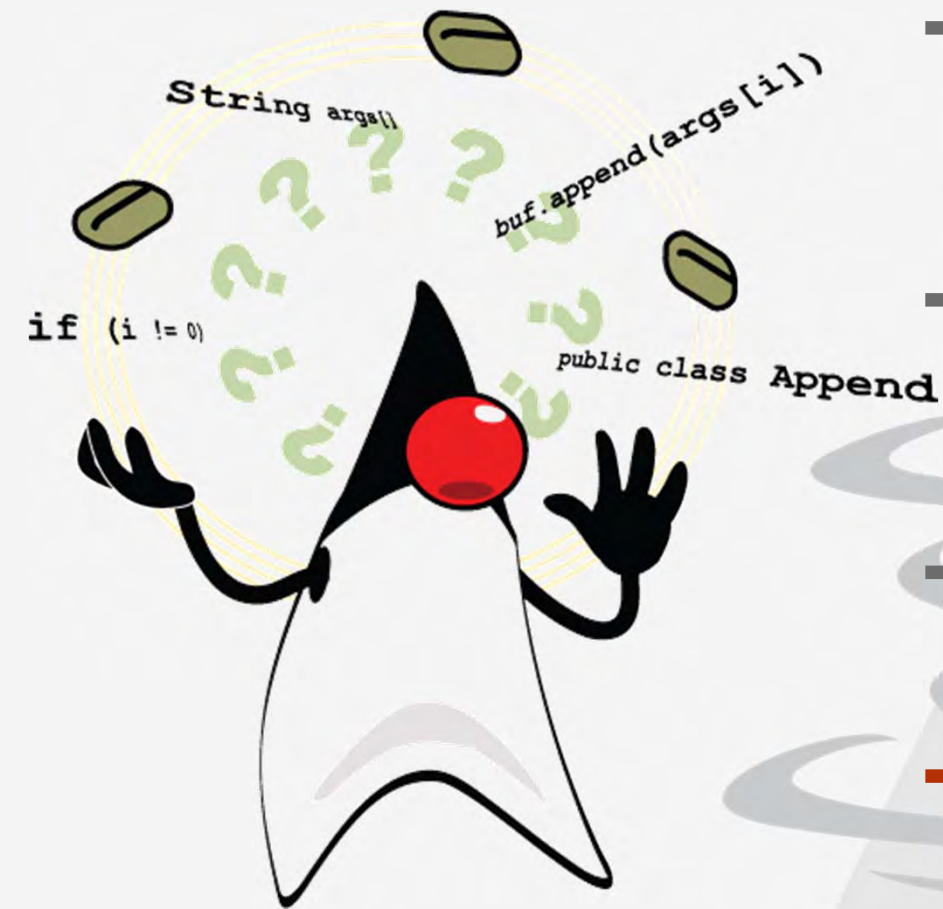
public, private, protected, default ✓

final, transient ✓

abstract, synchronized, strictfp, native

static → Variables de Clase

# Variables



- Las variables de clase son automáticamente inicializadas, las de método deben inicializarse específicamente.
- Una variable de clase es accesible desde cualquier punto de la clase, las de método son de acceso exclusivo en el método.
- Una variable de método puede llamarse igual que una de clase.
- El ámbito más cercano es el que prevalece, salvo cuando usamos el operador *this*.

# Arrays

## Arrays en Java

Array

2	23	a	dd	7a
0	1	2	3	4





# Arrays: Declaración

## Arrays en Java

→ De tipos primitivos u objetos

```
int numeros [ ];  
float [ ] decimales;  
Coches [ ] array_coches;  
Animal pets[ ];  
String [ ] [ ] palabras;
```





# Arrays: Declaración

## Arrays en Java

→ De tipos primitivos u objetos

`int numeros [5];`



# Arrays: Construcción

## Arrays en Java

- Se debe especificar el tamaño
- Se crean con el operador `new`

```
int[ ] testScores; // declaración  
testScores = new int[4]; // creación
```

```
// declaración + creación  
int[ ] testScores2 = new int[8];
```



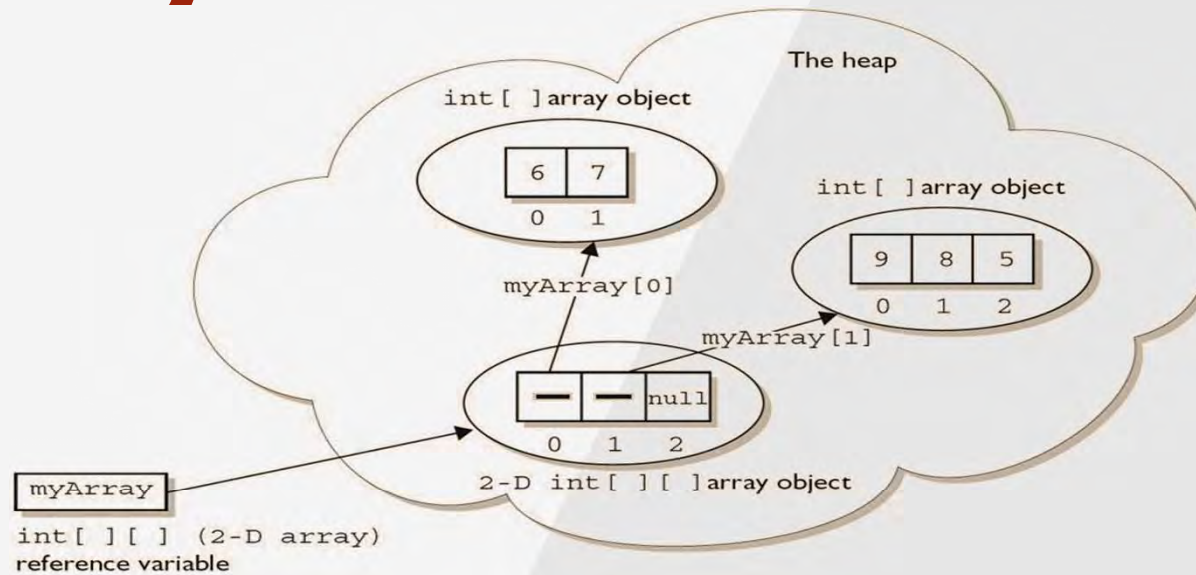
# Arrays: Multidimensión

## — Arrays en Java —

→ Son arrays que en cada posición, contienen otro array



# Arrays: Multidimensión



Picture demonstrates the result of the following code:

```
int[ ][ ] myArray = new int[3][ ];  
myArray[0] = new int[2];  
myArray[0][0] = 6;  
myArray[0][1] = 7;  
myArray[1] = new int[3];  
myArray[1][0] = 9;  
myArray[1][1] = 8;  
myArray[1][2] = 5;
```



# Arrays: Inicialización

## Arrays en Java

- Inicializar = Asignar valores
- Por defecto se crean sin valores
- El acceso a los elementos es por índice utilizando el [ ]
- El primer elemento es siempre el 0 y el último la longitud -1



# Arrays: Inicialización

## Arrays en Java

- Inicializar = Asignar valores
- Por defecto se crean sin valores
- El acceso a los elementos es por índice utilizando el [ ]

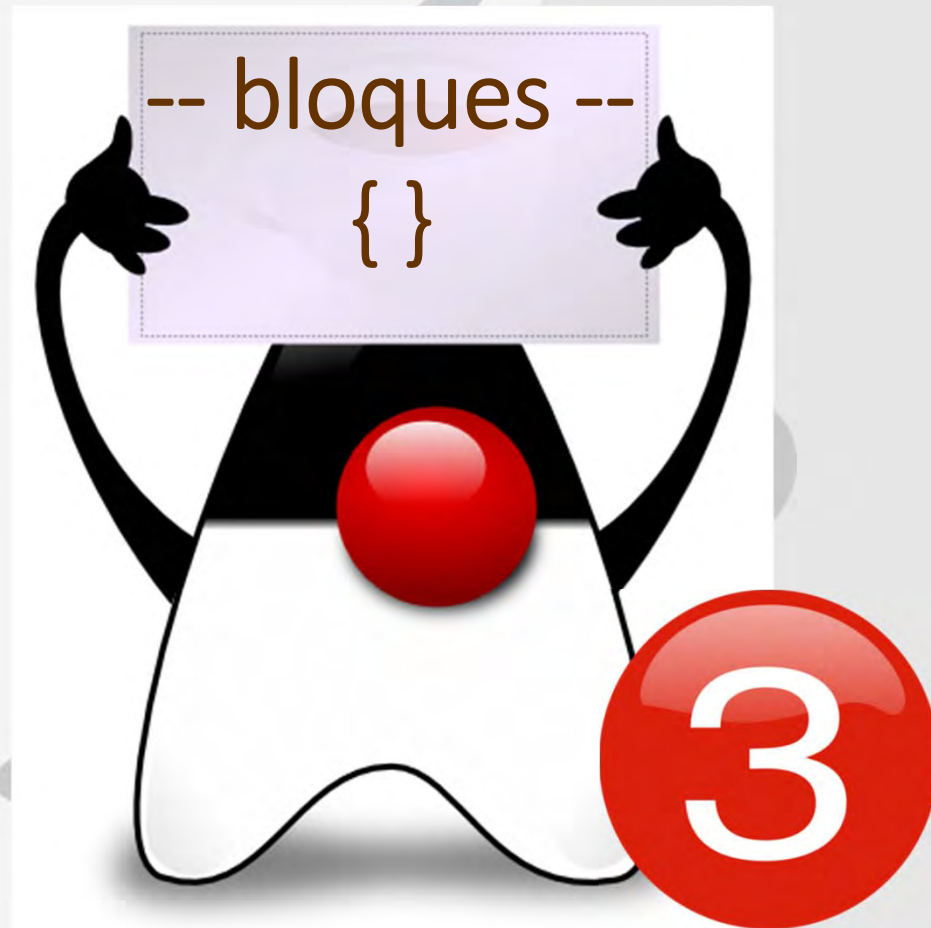


**ii ArrayIndexOutOfBoundsException !!**

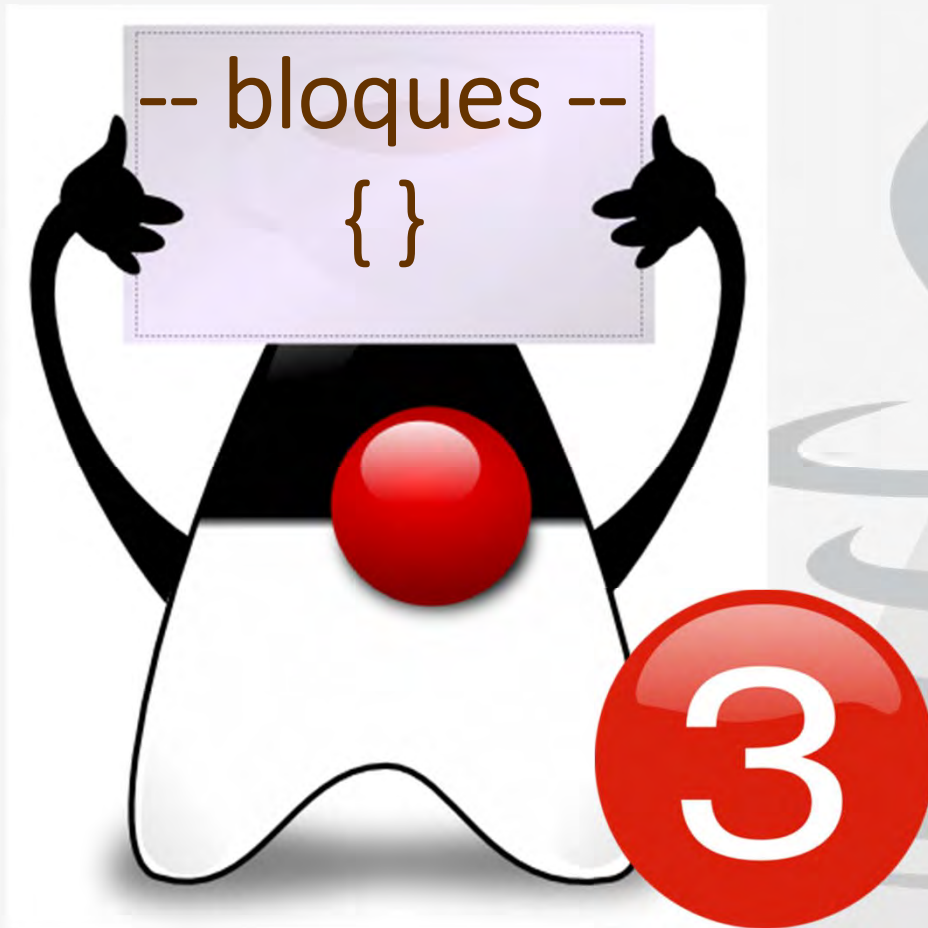
-- Error en tiempo de ejecución --



# Inicialización



# Inicialización



→ Constructores

→ Métodos

→ **Bloques**

→ Bloques de instancia {}

-- Después llamada a super --

→ **Bloques estáticos o de clase**  
**static {}**

-- Al cargar la clase --

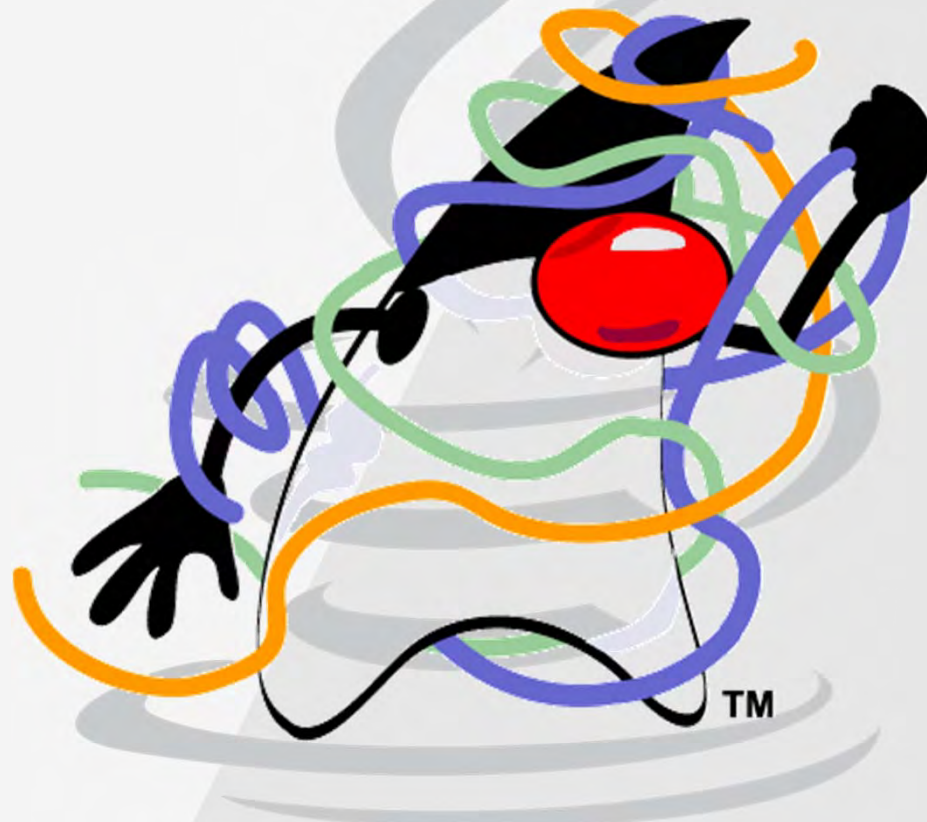
# Práctica: "Arrays"



# Práctica: "Arrays"

**Código que cree un array  
Y muestre sus valores...**

# Enums



# Enums

→ Son variables con valores restringidos a una lista

→ Se declaran en una clase propia o como miembro de clase





# Enums

→ Clase Propia



```
enum CoffeeSize { BIG, HUGE, OVERWHELMING }
```

```
class Coffee {  
    CoffeeSize size;  
}
```

```
public class CoffeeTest1 {  
    public static void main(String[] args) {  
        Coffee drink = new Coffee();  
        drink.size = CoffeeSize.BIG;  
    }  
}
```

# Enums

→ En otra Clase



```
class Coffee2 {  
    enum CoffeeSize {BIG, HUGE, OVERWHELMING }  
    CoffeeSize size;  
}  
  
public class CoffeeTest2 {  
    public static void main(String[] args) {  
        Coffee2 drink = new Coffee2();  
        drink.size = Coffee2.CoffeeSize.BIG;  
    }  
}
```

# Enums



```
public class CoffeeTest1 {  
    public static void main(String[] args) {  
        enum CoffeeSize { BIG, HUGE,  
OVERWHELMING }  
        Coffee d = new Coffee();  
        drink.size = Size.BIG;  
    }  
}
```



# Enums

→ Puede tener constructores



```
enum CoffeeSize {  
    BIG(8), HUGE(10), OVERWHELMING(16);
```

```
    CoffeeSize(int ounces)  
    {  
        this.ounces = ounces;
```

```
    public int getOunces()  
    {  
        return ounces;
```

```
    }
```



# Enums

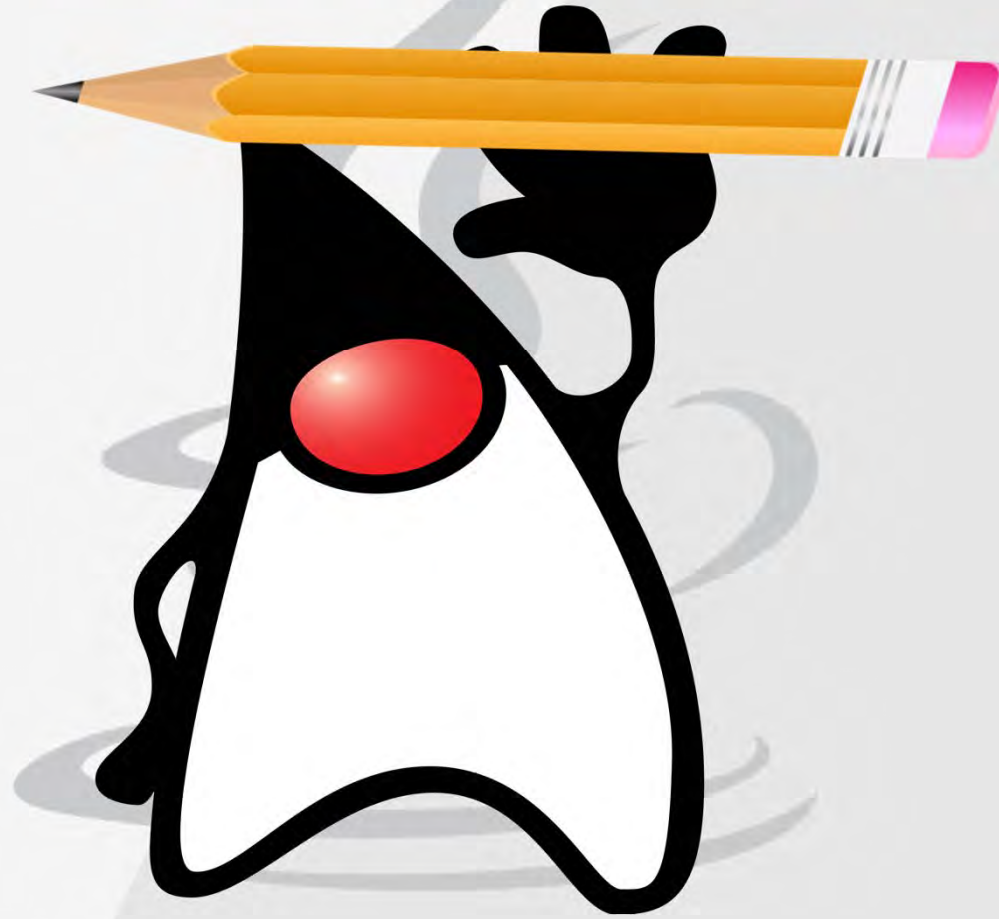


```
class Coffee {  
    CoffeeSize size;  
    public static void main(String[] args)  
    {  
        Coffee drink1 = new Coffee(CoffeeSize.BIG);  
        Coffee drink2 = new Coffee();  
        drink2.size = CoffeeSize.OVERWHELMING;  
        System.out.println(drink1.size.getOunces());  
        for(CoffeeSize cs: CoffeeSize.values()){  
            System.out.println(cs + " " +  
                               cs.getOunces());  
        }  
    }  
}
```





# Sobreescritura



# Sobreescritura



```
public class Animal
{
    public void comer()
    {
        System.out.println(" Animal
        Genérico comiendo");
    }
}
```

```
public class Caballo extends Animal
{
    public void comer()
    {
        System.out.println("Animal
        Caballo comiendo");
    }
}
```

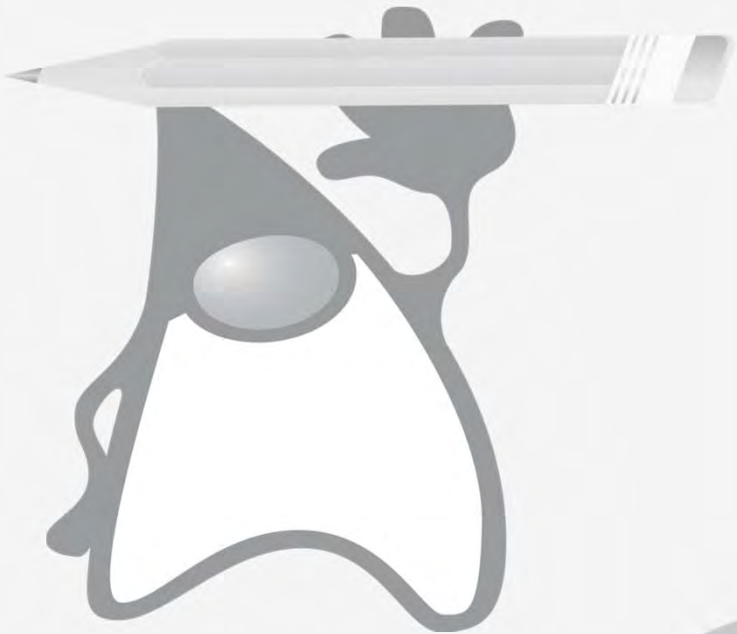
# Sobreescritura



- El argumento debe coincidir con el original.
- El tipo de retorno debe ser igual o subclase .
- El nivel de acceso no puede ser más restrictivo.
- No se pueden lanzar otras excepciones declaradas pero si no declaradas ( run –time ).
- Métodos estáticos o finales NO pueden ser sobreescritos.
- **Sólo métodos heredados pueden sobreescribirse.**

# Sobreescritura

→ Invocación al método original.



```
public class Animal {  
    public void comer() {}  
    public void correr()  
    }  
}  
  
class Caballo extends Animal {  
    public void correr() {  
        super.correr();  
    }  
}
```

# Sobreescritura

→ Invocación polimórfica.



```
class Animal {  
    public void comer() throws Exception {  
        // throws an Exception  
    }  
}  
class Perro extends Animal {  
    public void comer() {  
        // no Exceptions  
    }  
    public static void main(String [] args) {  
        Animal a = new Perro();  
        Perro p = new Perro();  
        p.comer();  
        a.comer();  
    }  
}
```

# Sobreescritura

→ Invocación polimórfica.



```
class Animal {  
    public void comer() throws Exception {  
        // throws an Exception  
    }  
}  
class Perro extends Animal {  
    public void comer() {  
        // no Exceptions  
    }  
    public static void main(String [] args) {  
        Animal a = new Perro();  
        Perro p = new Perro();  
        p.comer();  
        a.comer();  
    }  
}
```



# Sobrecarga



# Sobrecarga



```
class Animal {  
    public void comer()  
    {  
        // código comer generico  
    }  
}  
  
class Perro extends Animal {  
    public boolean comer (Integer unidades)  
    {  
        // código comer unidades  
    }  
    public void comer (Alimento [ ] alimentos)  
        throws IndigestionException  
    {  
        // código comer objetos Alimento  
    }  
}
```

# Sobrecarga



- Implica un cambio en la lista de argumentos de la firma del método.
- Puede cambiar el tipo de retorno.
- Puede cambiar el modificador de acceso.
- Puede cambiar las excepciones.
- Se puede dar a nivel de clase y de subclase.

# Test de Conocimientos

