

```

import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sck=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self, addr, port, func):
        try:
            self.sck.connect((addr, port))
            self.handle=self.sck
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error: could not connect"
    def listen(self, host, port, func):
        try:
            self.sck.bind((host, port))
            self.sck.listen(5)
            self.todo=1
            self.func=func
            self.start()
        except:
            print "Error: Could not bind"
    def run(self):
        while self.flag:
            if self.todo==1:
                x, ho=self.sck.accept()
                self.todo=2
                self.handle=x
            else:
                dat=self.handle.recv(4096)
                self.data=dat
                self.func()
    def send(self, data):
        self.handle.send(data)
    def close(self):
        self.flag=0
        self.sck.close()

```

DAM/DAW

PROGRAMACIÓN

06.1

OOP: Herencia en Java

Indice

- Introducción
- Herencia en Java
- Constructores y herencia
- Herencia y polimorfismo
- Clases abstractas
- El modificador *final*
- La clase Object

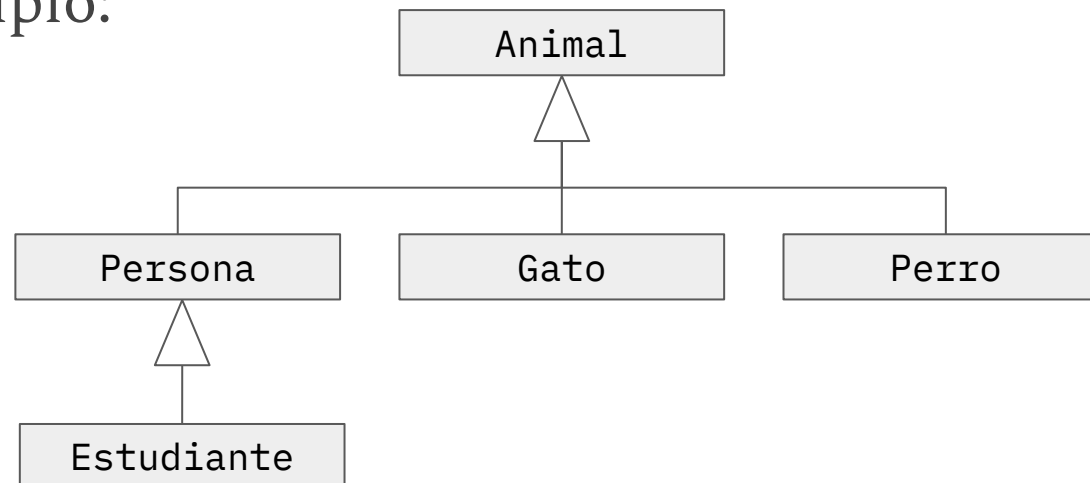
```
import threading, socket, time
class sock(threading.Thread):
    def __init__(self):
        self.sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        threading.Thread.__init__(self)
        self.flag=1
    def connect(self,addr,port,func):
        try:
            self.sock.connect((addr,port))
            self.handle=self.sock
            self.todo=2
            self.func=func
            self.start()
        except:
            print "Error:Could not connect"
            def listen(self,host,port,func):
                try:
                    self.sock.bind((host,port))
                    self.sock.listen(2)
                    self.todo=1
                    self.func=func
                    self.start()
                except:
                    print "Error:Could not bind"
            def run(self):
                while self.flag:
                    if self.todo==1:
                        x,ho=self.sock.accept()
                        self.todo=2
                        self.client=ho
                        self.handle=x
                    else:
                        dat=self.handle.recv(4096)
                        self.data=dat
                        self.func()
                def send(self, data):
                    self.handle.send(data)
            def close(self):
                self.flag=0
            self.sock.close()
```

Introducción

- La **herencia** es uno de los tres principios fundamentales de la programación orientada a objetos (junto a la encapsulación y el polimorfismo)
- La herencia nos permite crear **clases genéricas** que definen *rasgos* comunes para una serie de *entidades* del espacio del problema
- Esta clase “**padre**” o “**superclase**” podrá ser heredada por otras clases (“**subclases**”), incorporando las estructuras y comportamientos definidos por el padre. A su vez, la subclase podrá modificar dichos comportamientos o añadir los suyos propios.
- Una subclase es una **especialización** de la superclase. Hereda todas las variables y métodos definidos por el padre y añade sus propios elementos
- La herencia nos permite crear **jerarquías** de clases en las que, a medida que descendemos por la misma, se van refinando sus comportamientos (mayor especialización)

Herencia en Java (I)

- A diferencia de otros lenguajes, Java **no** soporta herencia múltiple. Una subclase sólo podrá tener una superclase (aparte de la clase *java.lang.Object* de la que, por defecto, derivan todas las clases en Java)
- Para indicar que una clase hereda de otra, añadiremos la palabra reservada ***extends*** en su declaración junto con el nombre de la superclase. Esto nos permitirá “incorporar” en la subclase el código de la clase padre (excepto miembros *default* si es una clase de otro paquete y *private*)
- Veamos un ejemplo:



Herencia en Java (II)

- Creamos la clase Animal:

```
public class Animal {  
    private int edad;  
    private String nombre;  
  
    public int getEdad() { return this.edad; }  
    public String getNombre() { return this.nombre; }  
    public void setEdad(int edad) { this.edad = edad; }  
    public void setNombre(String nombre) { this.nombre = nombre; }
```

@Override

```
public String toString() {  
    return "Animal:" + this.nombre + ":" + this.edad;
```

```
}  
}
```

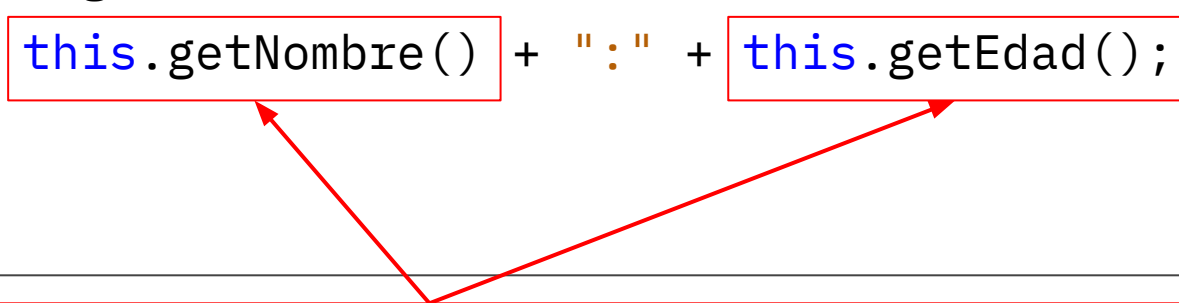
@Override: Anotación Java que fuerza al compilador a **comprobar** que se está sobrescribiendo un método de la clase padre (opcional)

método heredado
de
java.lang.Object

Herencia en Java (III)

- Ahora la clase Gato, que hereda de Animal...

```
public class Gato extends Animal {  
    public String habla() {  
        return "miau!!";  
    }  
  
    @Override  
    public String toString() {  
        return "Gato:" + this.getNombre() + ":" + this.getEdad();  
    }  
}
```



La clase *Gato* hereda los atributos **nombre** y **edad** de la clase *Animal*. Sin embargo, debido al modificador de acceso establecido para ellos (**private**), sólo podrá acceder a ellos mediante los métodos accesorios públicos (herados también)

Herencia en Java (IV)

- Vamos a crear una clase que nos permita probar la jerarquía...

```
public class Fauna {  
    public static void main(String[] args) {  
        Animal mi_animal = new Animal();  
        System.out.println(mi_animal);  
        mi_animal.setNombre("Dude");  
        mi_animal.setEdad(3);  
        System.out.println(mi_animal);  
        Gato yin = new Gato();  
        yin.setNombre("YinYang");  
        yin.setEdad(2);  
        System.out.println(yin);  
        System.out.println(yin.habla());  
    }  
}
```

Llamada al método *toString()* de la clase *Animal*

Llamadas a métodos accesorios definidos en la clase *Animal*

Se inicializan los atributos heredados de *Animal*

Llamadas a métodos accesorios heredados de la clase *Animal*

Llamada al método *toString()* sobreescrito de la clase *Gato*

Llamada a método propio de *Gato*

Herencia en Java (V)

- Y la salida será la siguiente...

```
Animal:null:0  
Animal:Dude:3  
Gato:YinYang:2  
miau!!
```

- Sigamos construyendo la jerarquía...

```
public class Perro extends Animal {  
    public String habla() {  
        return "guau!!";  
    }  
}
```

Como en el caso de Gato, la clase Perro heredará de Animal sus atributos y métodos, pero **sólo podrá acceder** a aquellos miembros que estén declarados como **public** o **protected**.

Podrá acceder a aquellos miembros que tengan establecido **acceso por defecto**, siempre y cuando la clase hija pertenezca al **mismo paquete** que su *superclase*.

Herencia en Java (y VI)

- Añadimo nuevo código a la clase de prueba...

```
public class Fauna {  
    public static void main(String[] args) {  
        ...  
        Perro blob = new Perro();  
        blob.setNombre("Blob");  
        blob.setEdad(5);  
        System.out.println(blob);  
        System.out.println(blob.habla());  
        ...  
    }  
}
```

Llamadas a métodos accesorios
definidos en la clase *Animal*

Llamada al método *toString()* de
Animal (*Perro* no tiene uno propio)

Llamada al método propio

```
...  
Animal:Blob:5  
guau!!
```

Constructores y herencia (I)

- En una jerarquía es posible, tanto para las superclases como las subclases, el definir sus propios constructores
- Como sabemos, al crear una nueva instancia de una clase, se invoca alguno de los constructores que tenga definidos (o el *constructor por defecto* si no tiene ninguno) con objeto de inicializar sus miembros.
- Ahora bien, cuando una clase hereda parte de sus atributos de otra clase superior, ¿quién se encarga de inicializar dichos atributos? ¿el constructor de la superclase o el de la subclase?
- En general, el constructor de la subclase invocará a un constructor de la superclase para inicializar los atributos heredados y, posteriormente, se encargará de inicializar el resto de los atributos
- Veamos con un ejemplo como se produce esta cadena de invocación de constructores...

Constructores y herencia (II)

- Añadimos a las clases `Animal` y `Gato` los siguientes constructores:

```
public class Animal {  
    public Animal() { System.out.println("> Constructor de Animal"); }  
    ...  
}  
  
public class Gato extends Animal {  
    public Gato() { System.out.println("> Constructor de Gato"); }  
    ...  
}  
  
public class Perro extends Animal {  
    public Perro() { System.out.println("> Constructor de Perro"); }  
    ...  
}
```

Constructores y herencia (III)

- Al ejecutar nuestra clase de prueba Fauna, obtendríamos los siguiente:

```
> Constructor de Animal
```

```
Animal:null:0
```

```
Animal:Dude:3
```

```
> Constructor de Animal
```

```
> Constructor de Gato
```

```
Gato:YinYang:2
```

```
miau!!
```

```
> Constructor de Animal
```

```
> Constructor de Perro
```

```
Animal:Blob:5
```

```
guau!!
```

Fíjate como la llamada al constructor del padre se ejecuta **antes** que cualquier otro código del constructor de la subclase

Al invocar a los constructores de las clases *Perro* y *Gato* se invoca **automáticamente** el constructor de la superclase *Animal*

Este constructor se encargará de **inicializar** los atributos que heredan las subclases (las subclases podrán modificar posteriormente esos valores si lo consideran necesario)

Constructores y herencia (IV)

- Acabamos de ver como el constructor de la *subclase*, lo primero que hace, es invocar automáticamente al constructor de la *superclase*
- Esto será posible **siempre que** no defina ningún constructor (en cuyo caso se usaría el constructor *por defecto* o *implícito*) o que alguno de sus constructores sea un constructor vacío (sin parámetros)
- Pero, ¿qué pasaría si **todos** los constructores que definiera la superclase fueran parametrizados? ¿qué constructor se invoca y con qué argumentos? La *subclase* **no** podrá hacerlo de forma automática...
- Para resolver esta situación, Java nos proporciona la construcción:

```
super(lista_de_parámetros);
```

- Mediante esta sentencia, el constructor de la *subclase* podrá invocar de forma **explícita** cualquier constructor de la *superclase*. La única condición es que sea la **primera** de las instrucciones del constructor de la *subclase*

Constructores y herencia (V)

- Vamos a añadir un constructor parametrizado a *Animal* :

```
public class Animal {  
    private int edad;  
    private String nombre;  
  
    public Animal() {  
        this("- sin nombre -", 0);  
    }  
  
    public Animal(String nombre, int edad) {  
        System.out.println("> Constructor de Animal");  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
}
```

Si se elimina el constructor sin parámetros, todas las *subclases* estarán obligadas a invocar explícitamente mediante *super()* el constructor parametrizado para inicializar los atributos heredados

Constructores y herencia (VI)

- Y modificamos los constructores de las clases *Gato* y *Perro*...

```
public class Gato extends Animal {  
    public Gato() {  
        System.out.println("> Constructor de Gato");  
    }  
  
    public Gato(String nombre, int edad) {  
        super(nombre, edad);  
        System.out.println("> Constructor de Gato");  
    }  
  
    ...  
}
```

Al no incluir una llamada explícita a `super()`, se invoca automáticamente el constructor sin parámetros de *Animal*

Llamada al constructor del padre
Tiene que ser la **primera** sentencia

Constructores y herencia (VIII)

- Continuamos la implementación de nuestra jerarquía...

```
public class Persona extends Animal {  
    protected boolean trabaja;  
    public Persona(String nombre, int edad) {  
        super(nombre, edad);  
        System.out.println("> Constructor de Persona");  
    }  
    public boolean getTrabaja() { return this.trabaja; }  
    public void setTrabaja(boolean trabaja) { this.trabaja = trabaja; }  
    public String habla() { return "hola!!"; }  
    @Override  
    public String toString() {  
        return "Persona:" + this.getNombre() + ":" + this.getEdad() +  
            ":" + this.trabaja;  
    }  
}
```

Invocamos el constructor del padre

Podemos acceder directamente a este atributo porque es propio de la clase

Constructores y herencia (VIII)

- ...y la clase Estudiante

```
import java.util.Random;

public class Estudiante extends Persona {
    public Estudiante(String nombre, int edad) {
        super(nombre, edad);
        this.trabaja = false;
        System.out.println("> Constructor de Estudiante");
    }

    public String habla() {
        Random r = new Random();
        return "Tengo " + ((r.nextInt(2) == 0) ? "hambre!" : "sueño!");
    }
}
```

Clase para generar números aleatorios

Invocamos el constructor del padre (*Persona*) que, a su vez, invocará el del "abuelo!!" (*Animal*)

Modificamos el atributo *trabaja* heredado de *Persona*
Al ser *protected*, podemos hacerlo directamente

Genera un número aleatorio en el rango $[0,2)$

Constructores y herencia (y IX)

- Bien! Terminamos de implementar nuestra jerarquía de clases!
- **Modifica** ahora la clase de prueba *Fauna* para crear distintas instancias de dichas clases y probar alguno de sus métodos
- Fíjate en lo siguiente:
 - Cómo se encadenan los constructores y en qué **orden**. Por ejemplo, al crear una instancia de la clase *Estudiante*, se completará primero la ejecución del constructor de *Animal*, luego el de *Persona* y, finalmente, el de *Estudiante*
 - Cada vez que invocamos un método, como por ejemplo *toString()* que se hereda desde la clase *Object* por toda la jerarquía hacia abajo, se busca si está sobrescrito en la clase. Si no, se busca en el padre. Si no, en la siguiente,... y así hacia arriba hasta encontrarlo

Herencia y polimorfismo (I)

- Como sabemos, Java es de tipado **estático** y **estricto**. Salvo contadas excepciones (promociones automáticas y conversiones), a una variable de tipo primitivo, **no** se le podrá asignar un valor de un tipo diferente al suyo
- Del mismo modo, una variable de tipo referencia de una clase, **no** podrá referenciar un objeto de otra clase
- Existe, sin embargo, una importante excepción de este último supuesto:

“A una variable de tipo referencia de una *superclase*, se le podrá asignar una referencia a un objeto de cualquiera de sus *subclases*”

- En todo caso, es importante entender que, es el ***tipo de la variable*** (no el ***tipo del objeto*** que referencia), lo que determina **qué miembros** pueden ser accedidos. Por tanto, cuando a una variable de tipo referencia de una *superclase* se le asigna un objeto de una de sus *subclases*:

“sólo podrá acceder a las partes del objeto definidas por la *superclase*”

Herencia y polimorfismo (II)

❖ Ejemplo (*jerarquía de clases*):

```
public class X {  
    int a;  
  
    public X(int a) { this.a = a; }  
}  
  
public class Y extends X {  
    int b;  
  
    public Y(int a, int b) {  
        super(a);  
        this.b = b; }  
}
```

Herencia y polimorfismo (III)

❖ Ejemplo (*clase de prueba*):

```
public class XYTest {  
    public static void main(String[] args) {  
        X x = new X(100);  
        X x2;  
        Y y = new Y(5, 6);  
  
        x2 = x;  
        System.out.println("x2.a = " + x2.a);  
        x2 = y;  
        System.out.println("x2.a = " + x2.a);  
  
        y.a = 13; // OK  
        x2.b = 23; // Error  
    }  
}
```

Una variable de la *superclase* puede referenciar a cualquier objeto de una de sus *subclases*

La *subclase* pueden acceder a los miembros heredados

La *superclase* no puede acceder a los miembros definidos por las *subclases*

Herencia y polimorfismo (IV)

❖ Otro ejemplo (nuevos constructores para *Persona* y *Estudiante*):

```
public class Persona extends Animal {  
    public Persona(Persona p) {  
        super(p.getNombre(), p.getEdad());  
        System.out.println("> Constructor de Persona");  
    }  
    ...  
}  
  
public class Estudiante extends Persona {  
    public Estudiante(Estudiante e) {  
        super(e);  
        this.trabaja = false;  
        System.out.println("> Constructor de Estudiante");  
    }  
    ...  
}
```

Crean un nuevos objetos a partir de instancias de la propia clase

Se invoca el constructor de la *superclase* **pero** pasando una instancia de la *subclase*

Herencia y polimorfismo (V)

- Cuando un método de una *subclase* tiene la misma *firma* (nombre y lista de parámetros) y devuelve el mismo valor que un método de su *superclase*, decimos que ese método está **sobreescrito**
- La **sobreescritura** de métodos en Java se implementa mediante un mecanismo denominado *Dynamic Method Dispatch*
- Este mecanismo facilita que un método sobreescrito sea resuelto en *tiempo de ejecución* en lugar de en *tiempo de compilación*
- La sobreescritura de métodos, junto con la capacidad de referenciar *subclases* desde variables del tipo de la *superclase*, facilitan la implementación de otro de los principios de la OOP: el **polimorfismo**

Cuando se invoca un método **sobreescrito** a través de la referencia de la **superclase**, Java determina qué método se ejecuta en base al **tipo del objeto** referenciado (no en base al **tipo de la variable** que lo referencia)

Herencia y polimorfismo (VI)

- Veamos un ejemplo de cómo funciona el **polimorfismo**...
- Primero añadiremos una implementación **genérica** del método *habla()* a la *superclase Animal*. El método será **sobreescrito** en cada una de sus *subclases*

```
public class Animal {  
    ...  
    public String habla() { return "Animal no habla!!"; }  
}
```

- El único objetivo de añadir este método a la clase *Animal*, es que toda la jerarquía (*superclase* incluida) presenten un **comportamiento común**. En realidad, se puede apreciar como implementar ciertas funcionalidades en una clase “genérica” como ésta puede resultar un tanto “absurdo”. De hecho, esta clase es una candidata perfecta para lo que en Java se conocen como **clases abstractas** (las veremos un poco más adelante)

Herencia y polimorfismo (VII)

- Vamos ahora a probar el polimorfismo desde nuestra clase de prueba...

```
public class Fauna {  
    public static void main(String[] args) {  
        ...  
        Animal[] lista = new Animal[3];  
        lista[0] = new Gato("Tom", 7);  
        lista[1] = new Perro("Scooby", 10);  
        lista[2] = new Estudiante("Rigby", 14);  
        for (Animal a: lista)  
            System.out.println(a.habla());  
    }  
}
```

Creamos una referencia a la *superclase*

Añadimos al *array* varias instancias de las distintas *subclases*

Invocamos el método definido en la *superclase* y sobrescrito en cada *subclase*

```
...  
miau!!  
guau!!  
Tengo hambre!
```

Observamos como, en cada caso, se ha empleado el *tipo del objeto* referenciado y no el *tipo de la variable*, para determinar *qué* implementación del método se ejecuta

Herencia y polimorfismo (VIII)

- Para finalizar con la sobrescritura de métodos, veamos cómo resolver aquellas situaciones en las que necesitamos que se ejecute la versión de la *superclase* del método sobrescrito
- Como sabemos, cuando una *subclase* sobrescribe un método de su *superclase*, la implementación de éste último queda “ocultada” por la versión de la *subclase*. Es decir, cada vez que invoquemos el método desde una instancia de la *subclase*, se ejecutará la versión definida por ella y no la de la *superclase*. Pero, ¿cómo podríamos acceder a la implementación del padre?
- Java nos ofrece una solución simple al problema. Del mismo modo que, cada vez que “entramos” en un método, *this* nos proporciona una referencia al objeto actual, *super* nos proporcionará una referencia a su *superclase* que podremos utilizar para acceder a sus métodos

Herencia y polimorfismo (y IX)

```
public class X {  
    int a;  
    public X(int a) { this.a = a; }  
    public void show() { System.out.println("a: " + this.a); }  
}  
  
public class Y extends X {  
    int b;  
    public Y(int a, int b) {  
        super(a);  
        this.b = b;  
    }  
    @Override  
    public void show() {  
        super.show();  
        System.out.println("b: " + this.b);  
    }  
}
```

Llamada al constructor de la *superclase*

Llamada a métodos de la *superclase*

Clases abstractas (I)

- En ocasiones desearemos crear una *superclase* que establezca una serie de comportamientos comunes para sus diferentes *subclases*, pero dejando toda la responsabilidad de la implementación a las mismas
- Lo acabamos de ver en la jerarquía del ejemplo anterior. Deseamos que todas las *subclases* de *Animal* dispongan de un método *habla()*, razón por la cual lo declaramos en la *superclase*. Sin embargo, la implementación del mismo debe realizarse en cada *subclase*, pues la forma particular de “hablar” de cada una ellas es diferente
- El problema nos lo encontramos al estar obligados a proporcionar una implementación para dicho método en la *superclase*, aún siendo conscientes de que carece de todo sentido.
- Es más, la clase *Animal* tiene sentido como *superclase* de la jerarquía pero, ¿realmente tiene sentido poder crear objetos de dicha clase?

Clases abstractas (II)

- Java nos proporciona el modificador *abstract* para dar respuesta a estas situaciones. Este modificador se aplica tanto a métodos como a clases
- **Un método abstracto no tendrá cuerpo** y, por tanto, no se implementará por la *superclase*. Todas las *subclases* están obligadas a sobrescribir dicho método, proporcionando una implementación para el mismo. Para declarar un método abstracto, emplearemos la siguiente sintaxis:

```
modif_acceso abstract tipo método(lista_parámetros);
```

- Una clase que contenga uno (o más) métodos abstractos, deberá ser **declarada como abstracta**. Para ello, añadiremos el modificador *abstract* antes de la palabra reservada *class* en la declaración de la clase
- **No se pueden crear instancias de una clase abstracta**
- Toda *subclase* de una clase abstracta debe implementar **todos** sus métodos abstractos. **De no hacerlo, debe ser declarada como abstracta**

Clases abstractas (y III)

- Finalmente, nuestra clase **abstracta** *Animal* quedará así:

```
public abstract class Animal {  
    private int edad;  
    private String nombre;
```

Al tener métodos abstractos, estamos obligados a declarar la clase como *abstract*. No se podrán crear objetos de esta clase

```
    public abstract String habla();
```

Método abstracto. No tiene cuerpo. Se implementa en las *subclases*

```
    public int getEdad() { return this.edad; }  
    public String getNombre() { return this.nombre; }  
    public void setEdad(int edad) { this.edad = edad; }  
    public void setNombre(String nombre) { this.nombre = nombre; }
```

```
@Override
```

```
    public String toString() {  
        return "Animal:" + this.nombre + ":" + this.edad;  
    }
```

La clase abstracta puede proporcionar a las *subclases* implementaciones genéricas de otros métodos no abstractos

```
}
```

El modificador *final*

- Habrá situaciones en las que deseamos prevenir que un método pueda ser sobrescrito por una clase heredada
- El modificador *final* que, como sabemos, aplicado a variables imposibilita que su valor sea modificado una vez la variable haya sido inicializada, puede ser empleado tanto con clases como con métodos. Para declarar un método final, emplearemos la siguiente sintaxis:

```
modif_acceso final tipo método(lista_parámetros);
```

Un **método** declarado como *final*, no podrá ser sobrescrito

- Podemos declara una clase como *final* añadiendo el modificador antes de la palabra reservada *class* en la declaración de la clase

Una **clase** declarada como *final*, no podrá ser heredada

- De lo anterior, se sobreentiende que una clase *final* no podrá se declarada como *abstract* y que todos sus métodos son, implícitamente, *final*

La clase *Object*

- Java define una clase especial denominada *Object* que es, de forma implícita, *superclase* de cualquier otra clase
- Esto implica que una variable de tipo *Object* podría referenciar a un objeto de cualquier clase
- Algunos métodos definidos por *Object* y que heredarán cualquier clase son:

Método	Descripción
<code>Object clone()</code>	Crea un nuevo objeto igual al objeto clonado
<code>boolean equals(Object obj)</code>	Determina si un objeto es igual a otro
<code>Class<?> getClass()</code>	Obtiene la clase del objeto
<code>int hashCode()</code>	Devuelve el <i>hash</i> asociado al objeto
<code>String toString()</code>	Devuelve una cadena que representa al objeto

- El método `getClass()` es *final* y no puede ser sobrescrito. Cualquiera de los otros métodos puede ser sobrescrito