

# Herencia Polimorfismo

# Herencia

Permite crear una **clase nueva** como un **subtipo de** una **clase existente**.

La **herencia** es una parte integral de Java, es el compilador el que hace la mayor parte del trabajo.

La **herencia es apropiada** solo cuando existe una **relación de subtipo genuina** entre la subclase y la superclase.

Es **seguro usar la herencia** dentro de un paquete, donde las **implementaciones de subclase y superclase** están bajo el control de los mismos programadores.

```
public class Circulo {  
    public static final double PI= 3.14159;  
    public double r;  
    public Circulo(double r) { this.r = r; }  
    public static double radianesAgrados(double rads) {  
        return rads * 180 / PI;  
    }  
    public double area() {  
        return PI * r * r;  
    }  
    public double circunferencia() {  
        return 2 * PI * r;  
    }  
}
```

```
public class CirculoPlano extends Circulo {  
    public double cx, cy;  
    public CirculoPlano(double r, double x, double y) {  
        super(r);  
        this.cx = x;  
        this.cy = y;  
    }  
    public boolean pertenece(double x, double y) {  
        double dx = x - cx, dy = y - cy;  
        double distancia = Math.sqrt(dx*dx + dy*dy);  
        return (distancia < r);  
    }  
}
```

Hereda de  
Circulo las v.i y  
los métodos

Invoca al constructor de la  
superclase Circulo(r)

El atributo r se hereda de  
Circulo y se usa en  
CirculoPlano como si fuese  
propio

# La Clase Object

Definición del método **equals()** de la clase Object:

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

Plaza plaza1 = **new** Plaza("Plaza Rocha");

Plaza plaza2 = **new** Plaza("Plaza Rocha");

System.out.println("Las plazas son iguales? "+ plaza1.equals(plaza2));

*¿Qué se imprime en pantalla?*      **Las plazas son iguales? false**

*¿Es el resultado esperado?*      **NO!! Esperamos true, dado que los valores de los objetos son iguales, ambos objetos representan a la plaza Rocha**

*¿Cuándo es apropiado sobrescribir el método **equals()** de **Object**?*

**Cuando las instancias de una clase tienen una noción de igualdad lógica que difiere de la identidad o referencias de esas instancias. Este es generalmente el caso de clases que representan valores, como por ejemplo las clases Integer, String de la API de JAVA.**

*¿Pasa lo mismo con las clases que tienen instancias controladas?* **NO**

# La Clase Object

El método **hashCode()** de la clase Object:

La sobrescritura del **hashCode()** está íntimamente relacionada con el framework de colecciones.

- Toda clase que sobrescribe el método **equals()** debe sobrescribir el **hashCode()** para asegurar un funcionamiento apropiado de sus objetos en contenedores basados en hashing como **HashMap**, **HashSet** y **HashTable**.
- Una correcta sobrescritura del método **hashCode()** asegura que **dos instancias lógicamente iguales tengan el mismo hashcode** y en consecuencia las estructuras de datos basadas en hashing que almacenan y recuperan estos objetos, funcionarán correcta y eficientemente.

Es una **buena práctica sobrescribir** el método **hashCode()** cuando se sobreescribe el **equals()** y de esta manera se respeta el contrato de la clase Object: *“si dos objetos son iguales según el método **equals()** entonces invocar al método **hashCode()** sobre cada uno de estos objetos, debe producir el mismo valor”*.

**ALWAYS OVERRIDE HASHCODE WHEN YOU OVERRIDE EQUALS**

**Effective JAVA** (está en la biblioteca) hay 2 ediciones **¿Cómo lo implemento?**

# La Clase Object

El método **toString()** de la clase Object:

- La versión original del método `toString()` definida en `Object` produce un string formado por el nombre de la clase, seguido del símbolo `@` más la representación hexadecimal del código hash del objeto sobre el que se invoca al método `toString()`.

`capitulo2.laplata.Plaza@adbf1`

- El objetivo de este método es producir una representación textual, concisa, legible y expresiva del contenido del objeto.

*¿Cuándo se invoca al método `toString()`?*

Cuando se pasa un objeto como parámetro en los métodos `print()`, `println()`, `printf()`, `println()`; con el operador de concatenación. Internamente las distintas versiones del método `print()` invocan al método `toString()`.

El método `toString()` también es usado por el Debugger, lo que facilita la interpretación de los pasos de la ejecución de un programa

*Es una buena práctica sobreescibir el método `toString()` para producir un resultado amigable que permite informar mejor sobre el objeto en cuestión.*

# ¿Cómo implementa JAVA la herencia?

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

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

```
public class Perro extends Mamifero{
    private String nom;
    public Perro(){
        System.out.println("Constructor de Perro");
    }
    public void setNombre(String n){this.nom=n;}
    public String getNombre(){return nom;}
    public void comer(){
        System.out.println("Perro.comer()");
    }
    public void jugar(){
        System.out.println("Perro.jugar()");
    }
    public static void main(String[] args){
        Perro p=new Perro();
    }
}
```

# Encadenamiento de Constructores

- Un **objeto de la clase derivada** incluye un **objeto de la clase base**.
- Es esencial que el **objeto de la clase base se inicialice correctamente**. La manera de garantizarlo es realizar la inicialización en el constructor **invocando al constructor de la clase base** quién tiene el conocimiento apropiado para hacerlo. Sólo el constructor de la clase base tiene el conocimiento y los accesos necesarios para inicializar a sus miembros.
- Es fundamental que sean invocados **todos los constructores de la cadena de herencia** y de esta manera garantizar que el **objeto** quede construido correctamente.
- El compilador Java silenciosamente invoca al constructor nulo o de *default* de la clase base en el constructor de la clase derivada (si no se lo invocó explícitamente). La invocación es automática.

## ¿Cómo se construye un objeto Perro?

Recorriendo la jerarquía de herencia en forma ascendente e invocando al constructor de la superclase desde cada nivel de la jerarquía de clases: el constructor de **Perro** invoca al constructor de **Mamifero**, el de **Mamifero** al de **Vertebrado** y el de **Vertebrado** al de **Object**.

La creación de un objeto Perro involucra:

- Ejecutar el cuerpo del constructor de Object -> crear un objeto de tipo Object
- Ejecutar el cuerpo del constructor Vertebrado -> crear un objeto de tipo Vertebrado
- Ejecutar el cuerpo del constructor Mamifero -> crear un objeto de tipo Mamifero
- Ejecutar el cuerpo del constructor Perro -> crear un objeto de tipo Perro

Si NO se codifica un constructor, el compilador insertará un constructor de default que invoca al constructor de default de la superclase



# Encadenamiento de Constructores

## Constructores con Argumentos

Para el compilador JAVA es sencillo invocar al constructor sin argumentos, sin embargo si la clase no tiene constructor nulo porque se escribió uno con argumentos o si se quiere invocar a un constructor con argumentos, la invocación debe hacerse en forma explícita usando la palabra clave **super** y la lista apropiada de argumentos.

```
public class Vertebrado{  
    private int cantpatas;  
    public Vertebrado(int patas){  
        System.out.println("Constructor de Vertebrado c/args");  
    }  
}
```

```
public class Mamifero extends Vertebrado{  
    public Mamifero(int patas){  
        super(patas);  
        System.out.println("Constructor de Mamifero c/args");  
    }  
}
```

La invocación al constructor de la clase base es lo primero que debe hacerse en el cuerpo del constructor de la clase derivada

```
public class Perro extends Mamifero{  
    private String nom;  
    public Perro(){  
        super(4);  
        System.out.println("Constructor de Perro");  
    }  
    public static void main(String[] args){  
        Perro p=new Perro();  
    }  
}
```

Si NO se invoca al constructor de la clase base explícitamente, el compilador insertará una invocación al constructor nulo. En nuestro ej. esta invocación implícita causaría un error de compilación.

Si una clase define constructores con argumentos y NO define al constructor nulo, todas sus subclasses deben definir constructores que invoquen explícitamente a los constructores con argumentos.



# Upcasting - Polimorfismo

Lo más interesante de la herencia es la relación entre la clase derivada y la clase base: “**la clase derivada es un tipo de la clase base**” (es-un o es-como-un). Java soporta esta relación.

**Upcasting** es la conversión de una referencia a un objeto de la clase derivada en una referencia a un objeto de la clase base.

**El upcasting es seguro:** la clase derivada es un super conjunto de la clase base, podría contener más métodos que la clase base, pero seguro contiene los métodos de la clase base.

*afinar(Nota)*

Vientos

Flauta

Oboe

Fagot

*Upcasting: Casting ascendente*

```
public class Musica{  
    public static void daleplay (Vientos v) {  
        v.afinar(Nota.DO);  
    }  
    public static void main(String[] args){  
        Flauta flauta=new Flauta();  
        Fagot fagot=new Fagot();  
        Oboe oboe=new Oboe();  
        dalePlay(flauta);  
        dalePlay(fagot);  
        dalePlay(oboe);  
    }  
}
```

*afinar() es polimórfico*

*dalePlay() acepta como parámetro una referencia a un objeto Vientos o a cualquier objeto derivado de Vientos*

*Una referencia a un objeto Flauta, Fagot y Oboe es pasada como parámetro al método dalePlay() sin hacer casting a Vientos*

*afinar(Nota) afinar(Nota) afinar(Nota)*

El **Upcasting** de Flauta, Oboe y Fagot a **Vientos** limita la interface pública de estos objetos a la de **Vientos**

# Polimorfismo ¿Cómo se implementa en Java?

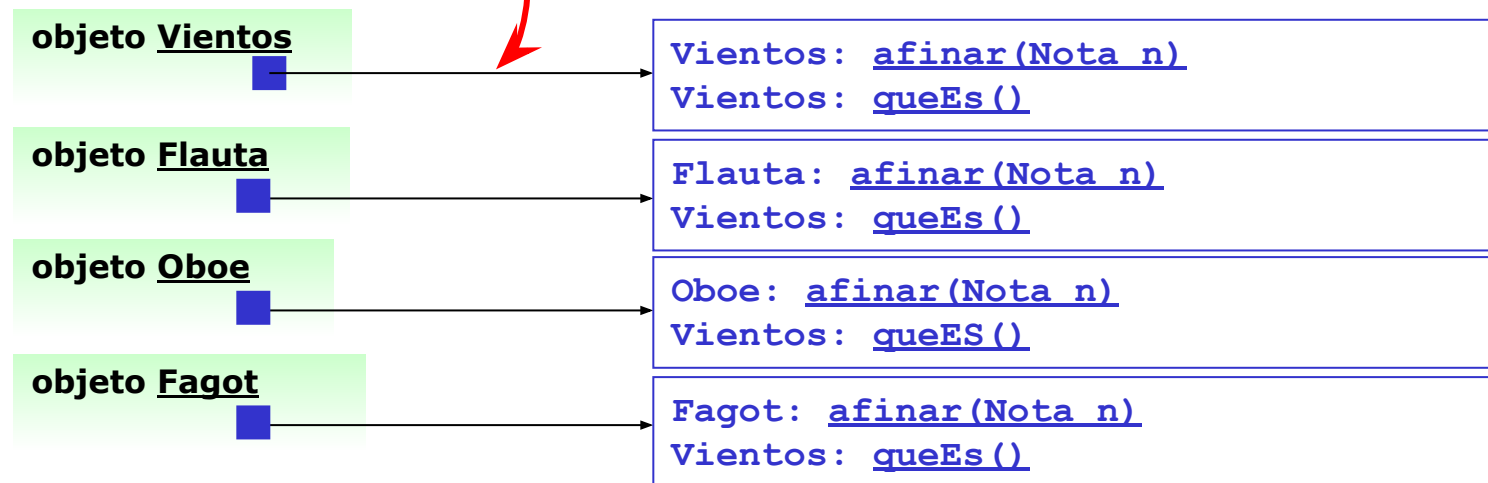
Cuando el “**intérprete**” ejecuta el código que produce el compilador busca el **afinar()** apropiado para invocar sobre cada objeto del arreglo. El intérprete chequea el tipo real del objeto referenciado por la variable y luego busca el método **afinar()** apropiado. El **intérprete** NO usa el método **afinar()** que está asociado estáticamente con la variable. **Dynamic Binding**

En JAVA la asociación entre la invocación a un método y el código que se ejecutará se resuelve **polimórficamente** a través del **binding dinámico**

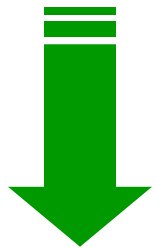
*Las invocaciones a afinar() y queEs() se resuelven mediante binding dinámico*

Objetos:

EJECUCIÓN



**Dynamic Binding**



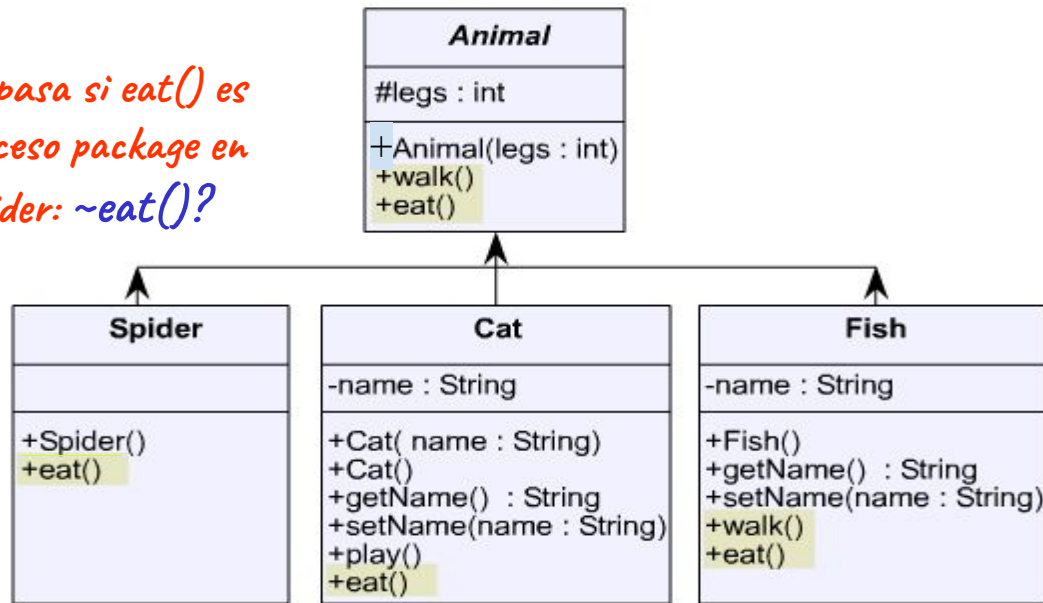
**Resuelve métodos polimórficos**

En Java el **Dynamic Binding** es automático. Todos los **métodos de una clase final**, todos los **métodos declarados final**, **private** o **static** son invocados sin usar **Dynamic Binding**. Las invocaciones a estos métodos son candidatas a ser optimizadas (por ej. usar *inlining*)

# eat()

# Sobreescritura y Control de Acceso

*¿Qué pasa si eat() es de acceso package en Spider: ~eat()?*



paquete1

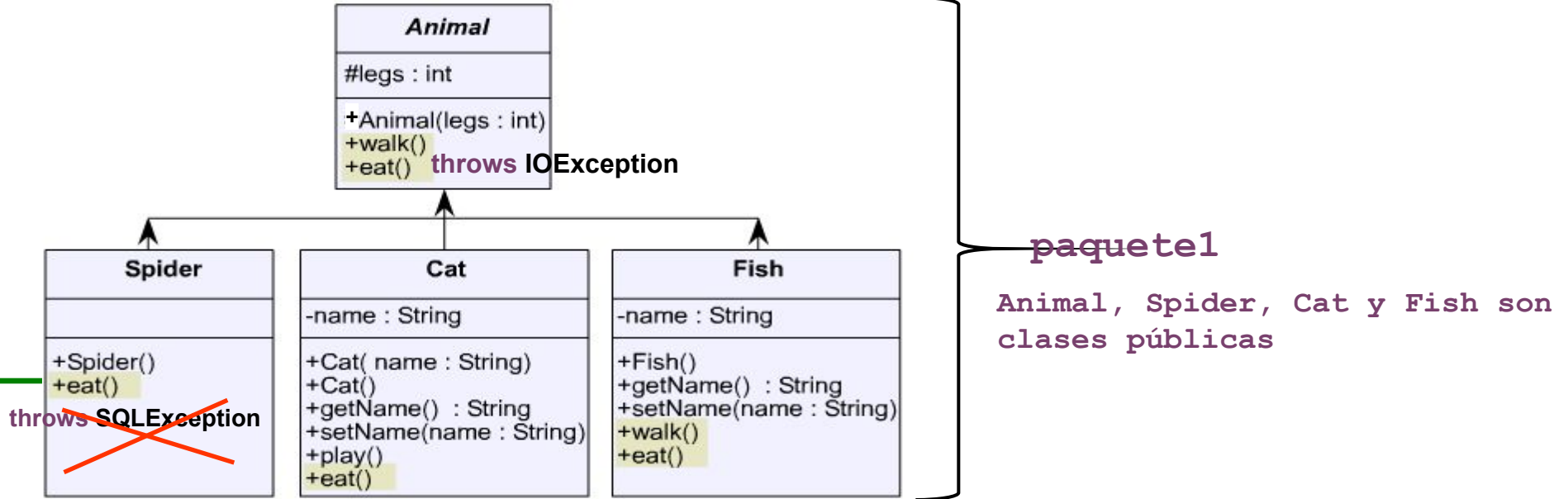
Animal, Spider, Cat y Fish son clases públicas

*eat()* es visible cuando se aplica sobre objetos Animal y deja de serlo cuando se aplica sobre una subclase (Spider), por lo tanto las subclases y las superclases dejan de tener una interface de comportamiento común

```
package paquete2;
import paquete1.*;
public class Fauna {
    private Animal[] ani={new Spider(), new Animal(4), new Cat("Violeta"), new Fish()};
    public static void main(String[] args) {
        Fauna f=new Fauna();
        for (int i=0;i<f.ani.length;i++){
            f.ani[i].eat();
        }
    }
}
```

Los métodos sobreescritos no pueden tener un control acceso más restrictivo que el declarado en la superclase

# Sobreescritura y Excepciones



```
package paquete2;
import paquete1.*;
import java.io.*;
public class Fauna {
    private Animal[] ani={new Spider(), new Animal(4), new Cat("Violeta"), new Fish()};
    public static void main(String[] args) throws IOException{
        Fauna f=new Fauna();
        for (int i=0;i<f.ani.length;i++){
            f.ani[i].eat();
        }
    }
}
```

Los métodos sobrescritos deben disparar las mismas excepciones que las del método de la superclase, subclases de ellas o ninguna excepción

# Reglas de Sobreescritura de Métodos

- Cualquier método que se herede (no privado y no final) de una superclase puede ser sobreescrito en las subclases.
- Los métodos sobreescritos en una subclase deben tener el mismo nombre, la misma lista de argumentos (en cuanto a tipo y orden) y el mismo tipo de retorno que los declarados en la superclase. El tipo de retorno es *covariante*, de esta manera el tipo de retorno puede ser una subclase del tipo de retorno del método original.
- El nivel de **acceso de un método sobreescrito** debe ser **igual o menos restrictivo** que el declarado en la **superclase**. Por ejemplo: si en la superclase el método es declarado **public** entonces el método sobreescrito en la subclase debe declararse **public**. Si en la superclase el método es declarado **protected** o **default**, en la subclase puede declararse **public**. ¿Y si es declarado **package**?
- Las **Excepciones** son clases especializadas que representan errores que pueden ocurrir durante la ejecución de un método. Los métodos sobreescritos deben disparar las mismas excepciones o subclases de las excepciones disparadas por el método original. NO pueden disparar otras excepciones.

# Constructores y Polimorfismo

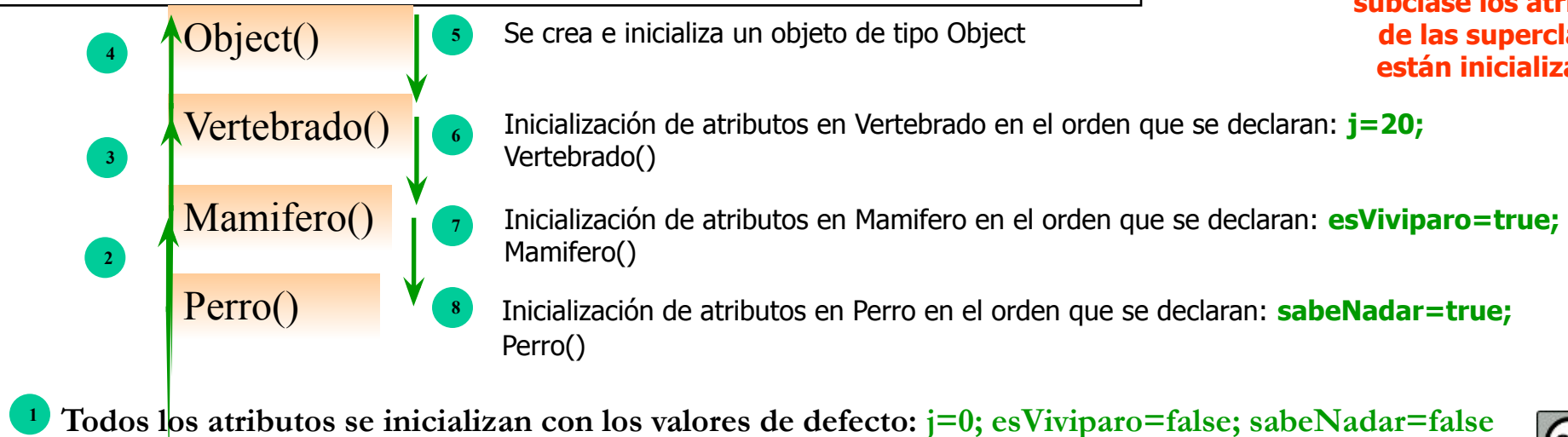
Los **constructores no son polimórficos** son implícitamente **static**.

```
public class Vertebrado {  
    private int j=20;  
  
    public Vertebrado() {  
        System.out.println("Vertebrado()");  
    }  
}
```

```
public class Mamifero extends Vertebrado {  
    private boolean esViviparo= true;  
    public Mamifero() {  
        System.out.println("Mamifero()");  
    }  
}
```

```
public class Perro extends Mamifero {  
    private boolean sabeNadar= true;  
  
    public Perro() {  
        System.out.println("Perro()");  
    }  
  
    public static void main(String[] args) {  
        Perro p=new Perro();  
    }  
}
```

Quando se ejecuta el constructor de una subclase los atributos de las superclases están inicializados



# Constructores y Polimorfismo

¿Qué ocurre si adentro de un constructor se invoca a un método sobrescrito?

```
public class Vertebrado {  
    private int j=20;  
  
    public Vertebrado() {  
        System.out.println("Vertebrado() antes de alimentar()");  
        alimentar();  
        System.out.println("Vertebrado() después de alimentar()");  
    }  
  
    public void alimentar() {  
        System.out.println("Vertebrado.alimentar()");  
    }  
}
```

```
public class Mamifero extends Vertebrado {  
    private boolean esViviparo= true;  
    public Mamifero() {  
        System.out.println("Mamifero()");  
    }  
    public void alimentar() {  
        System.out.println("Mamifero.alimentar(), vivíparo: "+ esViviparo);  
    }  
}
```

**j=20**

Vertebrado() antes de alimentar()

**Mamifero.alimentar(), esViviparo=false**

Vertebrado() después de alimentar()

```
public class Perro extends Mamifero {  
    private boolean sabeNadar= true;  
  
    public Perro() {  
        System.out.println("Perro()");  
    }  
  
    public static void main(String[] args) {  
        Perro p=new Perro();  
    }  
}
```

**esViviparo** aún NO se inicializó apropiadamente (de acuerdo a la declaración)

**j=0; esViviparo=false;**

**sabeNadar=false**

# Constructores y Polimorfismo

La invocación de constructores plantea un dilema :

¿Qué pasa si en el cuerpo de un constructor se invoca a un método del objeto que se está construyendo?

- Conceptualmente el constructor fabrica un objeto, pone a un objeto en un estado inicial.
- En el cuerpo de un constructor el objeto está parcialmente construido, sólo se sabe que los sub-objetos de las clases bases se inicializaron correctamente, pero no sabemos nada de las clases derivadas.
- Se estaría invocando a un método sobre un objeto que aún no se inicializó (su constructor NO se ejecutó), lo cual podría resultar en *bugs* difíciles de detectar, etc. Hay que tener en cuenta que el **binding dinámico** busca el método a invocar comenzando con la clase real del objeto y luego siguiendo por la cadena de herencia ascendentemente.

## Buena práctica para constructores:

- Los constructores **no deben invocar a métodos sobre-escribibles**.
- Los únicos métodos que son seguros para invocar en el cuerpo de un constructor son los declarados **final** en la clase base o **private** (que son automáticamente final). Estos métodos no pueden sobreescribirse y por lo tanto funcionan correctamente.



# Ocultar Atributos de la Superclase

```
public class Circulo {  
    public static final double PI= 3.14159;  
    public double r;  
    public Circulo(double r) { this.r = r; }  
    public static double radianesAgradados(double rads) {  
        return rads * 180 / PI;  
    }  
    public double area() {  
        return PI * r * r;  
    }  
    public double circunferencia() {  
        return 2 * PI * r;  
    }  
}
```

```
public class CirculoPlano extends Circulo {  
    public double r;  
    public double cx, cy;  
    public CirculoPlano(double r, double x, double y) {  
        super(r);  
        this.cx = x;  
        this.cy = y;  
        this.r = Math.sqrt(cx*cx + cy*cy);  
    }  
    public boolean pertenece(double x, double y) {  
        double dx = x - cx, dy = y - cy;  
        double distancia = Math.sqrt(dx*dx + dy*dy);  
        return (distancia < r);  
    }  
}
```

Se agrega a la clase **CirculoPlano** un atributo que guardará la distancia(**r**) entre el centro del círculo y el origen (0,0):

**r**

El atributo **r** de **CirculoPlano** **oculta** el atributo **r** de **Circulo**.  
Cuando en los métodos de **CirculoPlano**, nos referimos a:

**r** → es el atributo de **CirculoPlano**  
**this.r** → es el atributo de **CirculoPlano**  
**super.r** → es el atributo radio de **Circulo**: ((Circulo)this).r

¿A qué **r** hacen referencia los métodos **area()** y **circunferencia()** cuándo los invocamos sobre una instancia de **CirculoPlano**?

```
CirculoPlano cp=new CirculoPlano(5,10,10);  
cp.area();  
cp.circunferencia();
```

**Al radio definido en Circulo**



# Ocultar Atributos de la Superclase

```
public class Circulo {  
    public static final double PI= 3.14159;  
    public double r;  
    public Circulo(double r) { this.r = r; }  
    public static double radianesAgradados(double rads) {  
        return rads * 180 / PI;  
    }  
    public double area() {  
        return PI * r * r;  
    }  
    public double circunferencia() {  
        return 2 * PI * r;  
    }  
}
```

```
public class CirculoPlano extends Circulo {  
    public static final double PI = 3.14159265358979323846;  
    public double cx, cy;  
    public CirculoPlano(double r, double x, double y) {  
        super(r);  
        this.cx = x;  
        this.cy = y;  
    }  
    public boolean pertenece(double x, double y) {  
        double dx = x - cx, dy = y - cy;  
        double distancia = Math.sqrt(dx*dx + dy*dy);  
        return (distancia < r);  
    }  
}
```

¿Es posible **ocultar variables de Clase**? **SI !!**

Vamos a agregar una constante PI a **CirculoPlano**.

¿A qué PI hacen referencia area() y circunferencia() ?

**A la definida en Circulo, PI= 3.14159**

```
import static java.lang.System.out;  
public class TestOcultamiento {  
    public static void main(String args[]){  
        CirculoPlano cp=new CirculoPlano(10, 20, 10);  
        out.println("Area : " + cp.area());  
        out.println("Circunferencia: " + cp.circunferencia());  
    }  
    Area: 314.159  
    Circunferencia: 62.8318  
}
```

# Ocultar Métodos de Clase

Los métodos de clase de la misma manera que los atributos pueden **ocultarse** por una subclase, pero **NO sobrescribirse**. **NO son un reemplazo**.

```
class A {  
    int i = 1;  
    int f() { return i; }  
    static char g() { return 'A'; }  
}  
class B extends A {  
    int i = 2;  
    int f() { return -i; }  
    static char g() { return 'B'; }  
}
```

**Sobrescribe**      **Ocultar**

```
public class TestSobrescritura {  
    public static void main(String args[]) {  
        B b = new B();  
        System.out.println(b.i); // B.i; imprime 2  
        System.out.println(b.f()); // B.f(); imprime - 2  
        System.out.println(b.g()); // B.g(); imprime B  
        System.out.println(B.g()); // Imprime B  
                                   // Es la mejor manera de invocar a g()  
  
        A a = b; // Upcasting  
        System.out.println(a.i); // A.i; imprime 1 –OCULTA-  
        System.out.println(a.f()); // B.f(); imprime -2 –REEMPLAZA-  
        System.out.println(a.g()); // A.g(); imprime A –OCULTA-  
        System.out.println(A.g()); // Imprime A.  
                                   // Es la mejor manera de invocar a g()  
    }  
}
```