Java Básico

Herencia

Copyright

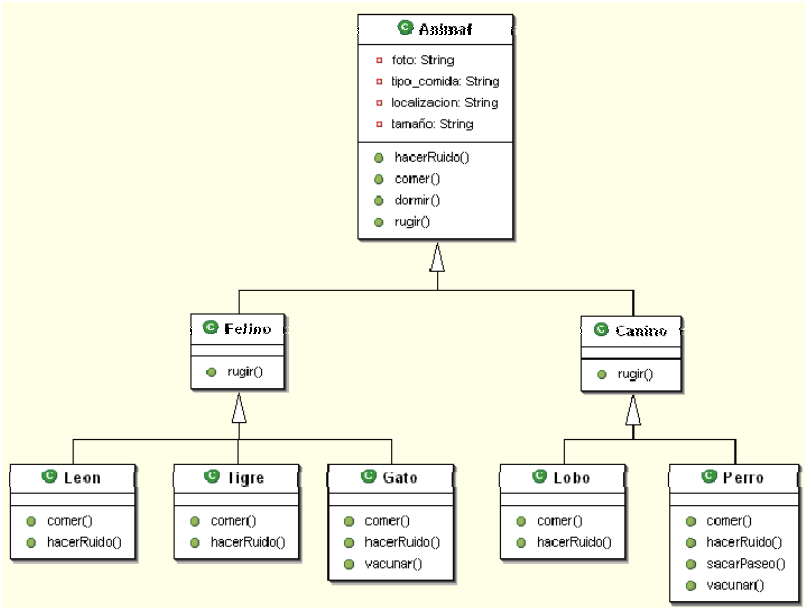
Relación de herencia

- Se basa en la existencia de relaciones de generalización/especialización entre clases.
- Las clases se disponen en una jerarquía, donde una clase hereda los atributos y métodos de las clases superiores en la jerarquía.
- Una clase puede tener sus propios atributos y métodos adicionales a lo heredado.
- Una clase puede modificar los atributos y métodos heredados.

Relación de herencia

- Las clases por encima en la jerarquía a una clase dada, se denominan superclases.
- Las clases por debajo en la jerarquía a una clase dada, se denominan subclases.
- Una clase puede ser superclase y subclase al mismo tiempo.
- Tipos de herencia:
  - Simple.
  - Múltiple (no soportada en Java)

Ejemplo



Herencia

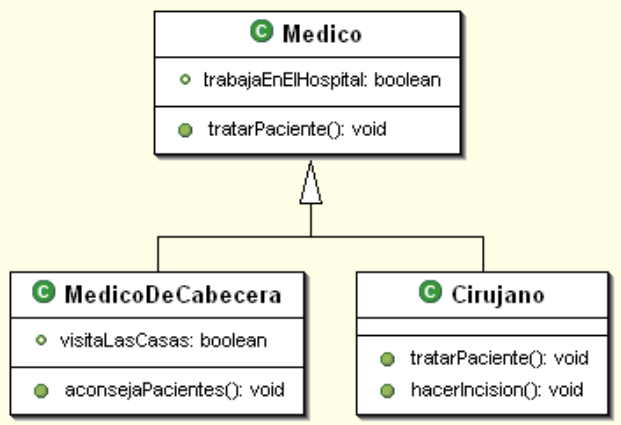
- La implementación de la herencia se realiza mediante la *keyword*: extends.
- Declaración de la herencia:

```
modificador_acceso class nom_clase extends nom_clase
{
}
```

- Ejemplo:

```
public class MiClase extends OtraClase
{
}
```

Ejemplo



Ejemplo

```
public class Medico
{
  public boolean trabajaEnHospital;

  public void tratarPaciente()
  {
    //Realizar un chequeo.
  }
}
```

```
public class MedicoDeCabecera extends Medico
{
  public boolean visitaLasCasas;

  public void aconsejaPacientes()
  {
    //Ofrecer remedios caseros.
  }
}

public class Cirujano extends Medico
{
  public void tratarPaciente()
  {
    //Realizar una operación.
  }

  public void hacerIncision()
  {
    //Realizar la incisión (¡ouch!).
  }
}
```

Ejercicio

- Contesta a las siguientes preguntas basándote en el ejemplo anterior:
  - ¿Cuántos atributos tiene la clase Cirujano?:__.
  - ¿Cuántos atributos tiene la clase MedicoDeCabecera?:__.
  - ¿Cuántos métodos tiene la clase Medico?:__.
  - ¿Cuántos métodos tiene la clase Cirujano?:__.
  - ¿Cuántos métodos tiene la clase MedicoDeCabecera?:__.
  - ¿Puede un MedicoDeCabecera tratar pacientes?:___.
  - ¿Puede un MedicoDeCabecera hacer incisiones?:___.

Ejercicio (solución)

- Contesta a las siguientes preguntas basándote en el ejemplo anterior:
  - ¿Cuántos atributos tiene la clase Cirujano?: 1.
  - ¿Cuántos atributos tiene la clase MedicoDeCabecera?: 2.
  - ¿Cuántos métodos tiene la clase Medico?: 1.
  - ¿Cuántos métodos tiene la clase Cirujano?: 2.
  - ¿Cuántos métodos tiene la clase MedicoDeCabecera?: 2.
  - ¿Puede un MedicoDeCabecera tratar pacientes?: Si.
  - ¿Puede un MedicoDeCabecera hacer incisiones?: No.

La clase Object

- En Java todas las clases heredan de otra clase:
  - Si lo especificamos en el código con la *keyword* extends, nuestra clase heredará de la clase especificada.
  - Si no lo especificamos en el código, el compilador hace que nuestra clase herede de la clase Object (raíz de la jerarquía de clases en Java).
- Ejemplo:

```
public class MiClase extends Object
{
   // Es redundante escribirlo puesto que el
   // compilador lo hará por nosotros.
}
```

La clase Object

- Esto significa que nuestras clases siempre van a contar con los atributos y métodos de la clase Object.
- Algunos de sus métodos mas importantes son:
  - **public boolean** equals(Object o);
    Compara dos objetos y dice si son iguales.
  - **public** String toString();
    Devuelve la representación visual de un objeto.
  - **public** Class getClass();
    Devuelve la clase de la cual es instancia el objeto.

La clase Object (cont.)

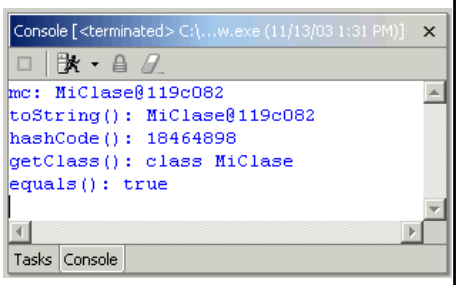
- **public int** hashCode();
  Devuelve un identificador unívoco después de aplicarle un algoritmo hash.
- **public** Object clone();
  Devuelve una copia del objeto.
- Otros métodos:
- **public void** finalize();
  Un método llamado por el Garbage Collector.
- **public void** wait(); **public void** notify();
  public void notifyAll();
  Tienen que ver con el manejo de threads.

Ejemplo

```
public class MiClase
{
}

public class TestMiClase
{
  public static void main(String[] args)
  {
    MiClase mc = new MiClase();

    System.out.println("mc: " + mc);
    System.out.println("toString(): " + mc.toString());
    System.out.println("hashCode(): " + mc.hashCode());
    System.out.println("getClass(): " + mc.getClass());
    System.out.println("equals(): " + mc.equals(mc));
  }
}
```



Castings

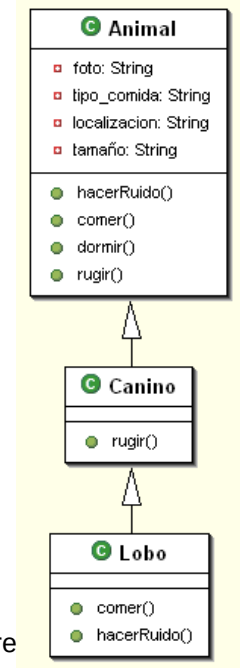
- El casting es una forma de realizar conversiones de tipos.
- Hay dos clases de casting:
  - UpCasting: conversión de un tipo en otro superior en la jerarquía de clases. No hace falta especificarlo.
  - DownCasting: conversión de un tipo en otro inferior en la jerarquía de clases.
- Se especifica precediendo al objeto a convertir con el nuevo tipo entre paréntesis.

Ejemplo

```
public class Test
{
  public static void main (String[] args)
  {
    Lobo lobo = new Lobo();

    // UpCastings
    Canino canino = lobo;
    Object animal = new Lobo();
    animal.comer();

    // DownCastings
    lobo = (Lobo)animal;
    lobo.comer();
    Lobo otroLobo = (Lobo)canino;
    Lobo error = (Lobo) new Canino();
  }
```



No compila. No puedes llamar al método comer() sobre un Object. No puedes convertir un Canino en un Lobo.

Sobrescribir un método

- Sobrescribir un método significa que una subclase reimplementa un método heredado.
- Para sobrescribir un método hay que respetar totalmente la declaración del método:
  - El nombre ha de ser el mismo.
  - Los parámetros y tipo de retorno han de ser los mismos.
  - El modificador de acceso no puede ser mas restrictivo.
- Al ejecutar un método, se busca su implementación de abajo hacia arriba en la jerarquía de clases.

Ejemplo

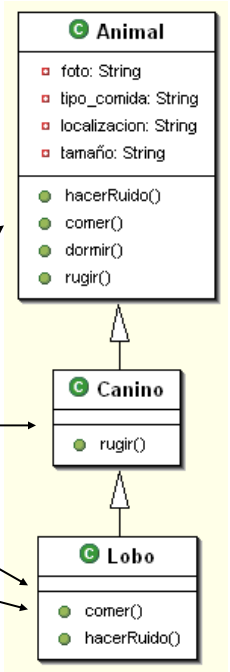
```
public class Test
{
  public static void main (String[] args)
  {
    Lobo lobo = new Lobo();

    lobo.hacerRuido();

    lobo.rugir();

    lobo.comer();

    lobo.dormir();
  }
```



Ejemplo

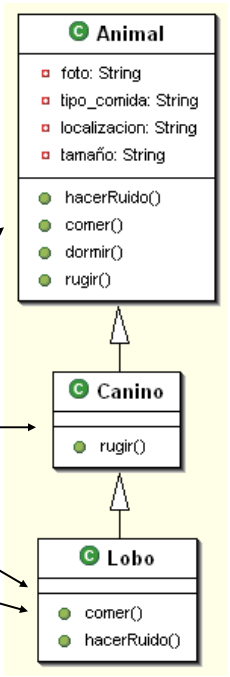
```
public class Test
{
  public static void main (String[] args)
  {
    Animal animal = new Lobo();

    animal.hacerRuido();

    animal.rugir();

    animal.comer();

    animal.dormir();
  }
```



Los castings no modifican al objeto. Solo su tipo, por lo que se siguen ejecutando sobre el mismo objeto.

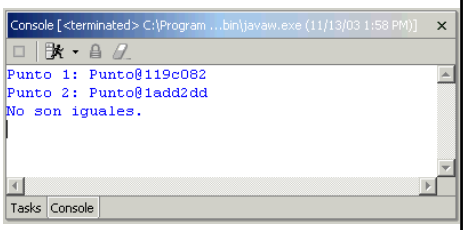
Ejemplo

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

  public Punto(int param1, int param2)
  {
    x = param1;
    y = param2;
  }
}

public class Test
{
  public static void main(String[] args)
  {
    Punto pun1 = new Punto(1,2);
    Punto pun2 = new Punto(1,2);
    System.out.println("Punto 1: " + pun1);
    System.out.println("Punto 2: " + pun2);
    if(pun1.equals(pun2))
      System.out.println("Son iguales.");
  }
}
```

Se ha llamado al método equals() de la clase Object que no sabe nada sobre cuando dos puntos son iguales matemáticamente hablando. El método toString() tampoco sabe como mostrar información de un punto por pantalla.

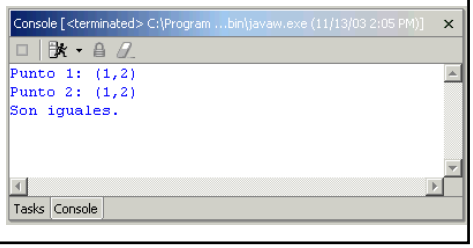


Ejemplo

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

  public Punto(int param1, int param2)
  {
    x = param1;
    y = param2;
  }
  public String toString()
  {
    return "(" + x + "," + y + ")";
  }
  public boolean equals(Object o)
  {
    Punto p = (Punto)o;
    if(x == p.x && y == p.y)
      return true;
    else
      return false;
  }
}
```

```
public class Test
{
  public static void main(String[] args)
  {
    Punto pun1 = new Punto(1,2);
    Punto pun2 = new Punto(1,2);
    System.out.println("Punto 1: " + pun1);
    System.out.println("Punto 2: " + pun2);
    if(pun1.equals(pun2))
      System.out.println("Son iguales.");
  }
}
```



Sobrescribir vs. Sobrecargar

- Sobrecargar un método es un concepto distinto a sobrescribir un método.
- La sobrecarga de un método significa tener varias implementaciones del mismo método con parámetros distintos:
  - El nombre ha de ser el mismo.
  - El tipo de retorno ha de ser el mismo.
  - Los parámetros tienen que ser distintos.
  - El modificador de acceso puede ser distinto.

Sobrescribir vs. Sobrecargar

- Habrá que tener muy en cuenta los parámetros que se envían y las conversiones por defecto para saber qué método se ejecuta.
- Por ejemplo:
  - Tenemos un método que recibe un float.

    public void miMetodo(float param) { }
  - miObjeto.miMetodo(1.3); llamará sin problemas al método.
  - Sobrecargamos el método para que reciba un double.

    public void miMetodo(double param) { }

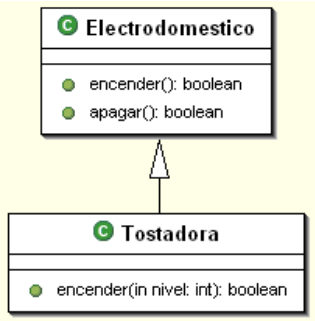
Sobrescribir vs. Sobrecargar

- Continuación del ejemplo:
  - miObjeto.miMetodo(1.3); ya no llama al método con float.
  - Recordemos que un número real por defecto es double.
  - Para seguir llamando al método con float debemos especificarlo implícitamente:
  - miObjeto.miMetodo(1.3F); o miObjeto.miMetodo((float)1.3);

Ejemplo

```
public class Electrodomestico
{
  public boolean encender()
  {
    //Hacer algo.
  }
  public boolean apagar()
  {
    //Hacer algo.
  }
}

public class Tostadora extends Electrodomestico
{
  public boolean encender(int nivel)
  {
    //Hacer algo.
  }
}
```

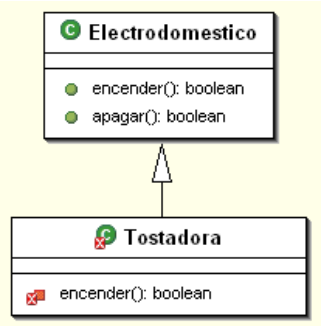


No es sobrescritura. Los parámetros son distintos. Es sobrecarga.

Ejemplo

```
public class Electrodomestico
{
  public boolean encender()
  {
    //Hacer algo.
  }
  public boolean apagar()
  {
    //Hacer algo.
  }
}

public class Tostadora extends Electrodomestico
{
  private boolean encender()
  {
    //Hacer algo.
  }
}
```



No compila. Es sobrescritura restringiendo el acceso.

Sobrecarga de métodos

- Java SE 5.0 añade una novedad al respecto.
- Se permite la sobrecarga de métodos cambiando también el tipo de retorno, pero siempre que:
  - El método que se está sobrecargando sea de una clase padre (de la que heredamos directa o indirectamente).
  - El nuevo tipo de retorno sea hijo del tipo de retorno del método original (es decir, que herede de él directa o indirectamente).
- Por tanto, no es válido para tipos primitivos.
- Veremos con mas detalle esta nueva característica en el capítulo dedicado a Java SE 5.0

El uso de la Herencia

- Debemos usar herencia cuando hay una clase de un tipo mas específico que una superclase.
  Es decir, se trata de una especialización.
  - Lobo es mas específico que Canino. Luego tiene sentido que Lobo herede de Canino.
- Debemos usar herencia cuando tengamos un comportamiento que se puede reutilizar entre varias otras clases del mismo tipo genérico.
  - Las clases Cuadrado, Circulo y Triangulo tiene que calcular su área y perímetro luego tiene sentido poner esa funcionalidad en una clase genérica como Figura.

El uso de la Herencia

- No debemos usar herencia solo por el hecho de reutilizar código. Nunca debemos romper las dos primeras reglas.
  - Podemos tener el comportamiento cerrar en Puerta. Pero aunque necesitemos ese mismo comportamiento en Coche no vamos a hacer que Coche herede de Puerta. En todo caso, coche tendrá un atributo del tipo Puerta.
- No debemos usar herencia cuando no se cumpla la regla: Es-un (Is-a).
  - Refresco es una Bebida. La herencia puede tener sentido. Bebida es un Refresco. ¿? No encaja luego la herencia no tiene sentido.

Ejercicio

```
public class A
{
  int ivar = 7;
  public void m1()
  {
    System.out.println("A's m1, ");
  }
  public void m2()
  {
    System.out.println("A's m2, ");
  }
 public void m3()
  {
    System.out.println("A's m3, ");
  }
}

public class B extends A
{
 public void m1()
  {
    System.out.println("B's m1, ");
  }
}

public class C extends B
{
 public void m3()
  {
    System.out.println("C's m3, " + (ivar + 6));
  }
}

public class Mix
{
  public static void main(String[] args)
  {
    A a = new A();
    B b = new B();
    C c = new C();
    A a2 = new C();
```

¿?

```
  }
}
```

Ejercicio

- ¿Qué salida produce la inclusión en el programa anterior de estas tres líneas de código en el recuadro vació?

```
b.m1();
c.m2();
a.m3();

c.m1();
c.m2();
c.m3();

a.m1();
b.m2();
c.m3();

a2.m1();
a2.m2();
a2.m3();
```

A's m1, A's m2, C's m2, 6

B's m1, A's m2, A's m3,

A's m1, B's m2, A's m3,

B's m1, A's m2, C's m3, 13

B's m1, C's m2, A's m3,

B's m1, A's m2, C's m3, 6

A's m1, A's m2, C's m3, 13

Solución

```
b.m1();
c.m2();
a.m3();

c.m1();
c.m2();
c.m3();

a.m1();
b.m2();
c.m3();

a2.m1();
a2.m2();
a2.m3();
```

A's m1, A's m2, C's m2, 6

B's m1, A's m2, A's m3,

A's m1, B's m2, A's m3,

B's m1, A's m2, C's m3, 13

B's m1, C's m2, A's m3,

B's m1, A's m2, C's m3, 6

A's m1, A's m2, C's m3, 13

super y this

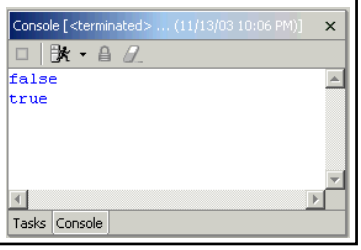
- super y this son dos *keywords* de Java.
- super es una referencia al objeto actual pero apuntando al padre.
- super se utiliza para acceder desde un objeto a atributos y métodos (incluyendo constructores) del padre.
- Cuando el atributo o método al que accedemos no ha sido sobrescrito en la subclase, el uso de super es redundante.
- Los constructores de las subclases incluyen una llamada a super() si no existe un super o un this.

super y this

- Ejemplo de acceso a un atributo:

```
public class ClasePadre
{
  public boolean atributo = true;
}


public class ClaseHija extends ClasePadre
{
  public boolean atributo = false;
  public void imprimir()
  {
    System.out.println(atributo);
    System.out.println(super.atributo);
  }
}
```

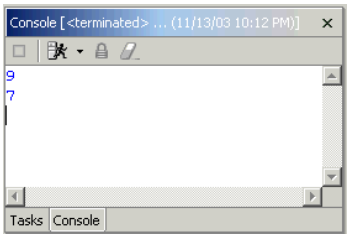


super y this

- Ejemplo de acceso a un constructor:

```
public class ClasePadre
{
  public ClasePadre(int param)
  {
    System.out.println(param);
  }
}
public class ClaseHija extends ClasePadre
{
  public ClaseHija(int param)
  {
    super(param + 2);
    System.out.println(param);
  }
}
```

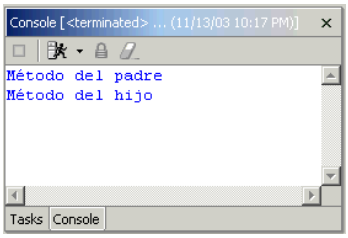
Nota: tiene que ser la primera línea del constructor y solo puede usarse una vez por constructor.



super y this

- Ejemplo de acceso a un método:

```
public class ClasePadre
{
  public void imprimir()
  {
    System.out.println("Método del padre");
  }
}
public class ClaseHija extends ClasePadre
{
  public void imprimir()
  {
    super.imprimir();
    System.out.println("Método del hijo");
  }
}
```



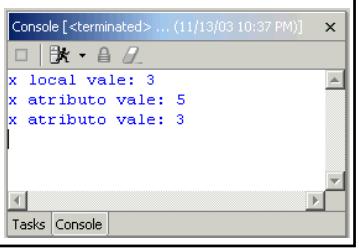
super y this

- this es una referencia al objeto actual.
- this se utiliza para acceder desde un objeto a atributos y métodos (incluyendo constructores) del propio objeto.
- Existen dos ocasiones en las que su uso no es redundante:
  - Acceso a un constructor desde otro constructor.
  - Acceso a un atributo desde un método donde hay definida una variable local con el mismo nombre que el atributo.

super y this

- Ejemplo de acceso a un atributo:

```
public class MiClase
{
  private int x = 5;
  public void setX(int x)
  {
    System.out.println("x local vale: " + x);
    System.out.println("x atributo vale: " + this.x);
    this.x = x;
    System.out.println("x atributo vale: "
                                        + this.x);
  }
}
```

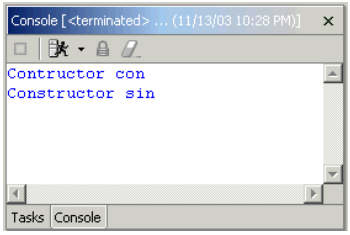


super y this

- Ejemplo de acceso a un constructor:

```
public class MiClase
{
  public MiClase()
  {
    this(2);
    System.out.println("Constructor sin");
  }
  public MiClase(int param)
  {
    System.out.println("Contructor con");
  }
}
```

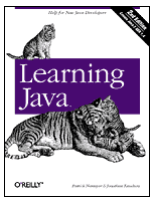
Nota: tiene que ser la primera línea del constructor y solo puede usarse una vez por constructor.



Bibliografía

- Head First Java (2nd edition)
  Kathy Sierra y Bert Bates.
  O'Reilly

- Learning Java (2nd edition)
  Patrick Niemeyer y Jonathan Knudsen.
  O'Reilly.



- Thinking in Java (4th edition)
  Bruce Eckel.
  Prentice Hall.

- The Java tutorial
  http://java.sun.com/docs/books/tutorial/