



UNIVERSIDAD DE ALMERÍA

Grado en Ingeniería Informática

Metodología de la Programación

2016-2017



Tema 1. Herencia y polimorfismo

- Superclase y subclase
- Polimorfismo
- Enlace dinámico
- Casting de objetos y el operador instanceof
- El método de objeto equals
- Propiedades y métodos protected
- El modificador final
- Resumen



La programación orientada a objetos permite definir nuevas clases a partir de otras que ya existen. Esto se llama **herencia**.

La **herencia** es una característica muy importante y poderosa que permite reusar software.

Supongamos que necesitamos definir clases para modelar círculos, rectángulos y triángulos. Estas clases tienen características comunes ¿Cuál es el mejor camino en el diseño de estas clases para evitar redundancias y hacer el sistema más fácil de entender y de mantener? La respuesta es usar herencia.

Superclase y Subclase

La herencia permite definir una clase general (superclase) y después extender esta a clases más especializadas (subclase).

Usamos una clase para modelar objetos del mismo tipo. Distintas clases pueden tener características y comportamientos comunes que pueden generalizarse en una clase que pueden compartir otras clases. Se puede definir una clase especializada que amplía la clase generalizada. Las clases especializadas heredan las propiedades y métodos de la clase general.

Supongamos que queremos diseñar clases para modelar objetos geométricos tales como círculos y rectángulos. Los objetos geométricos tienen propiedades comunes tales como el **color**, **relleno** y **fechaCreado** y los métodos **get**, **set** y **toString**

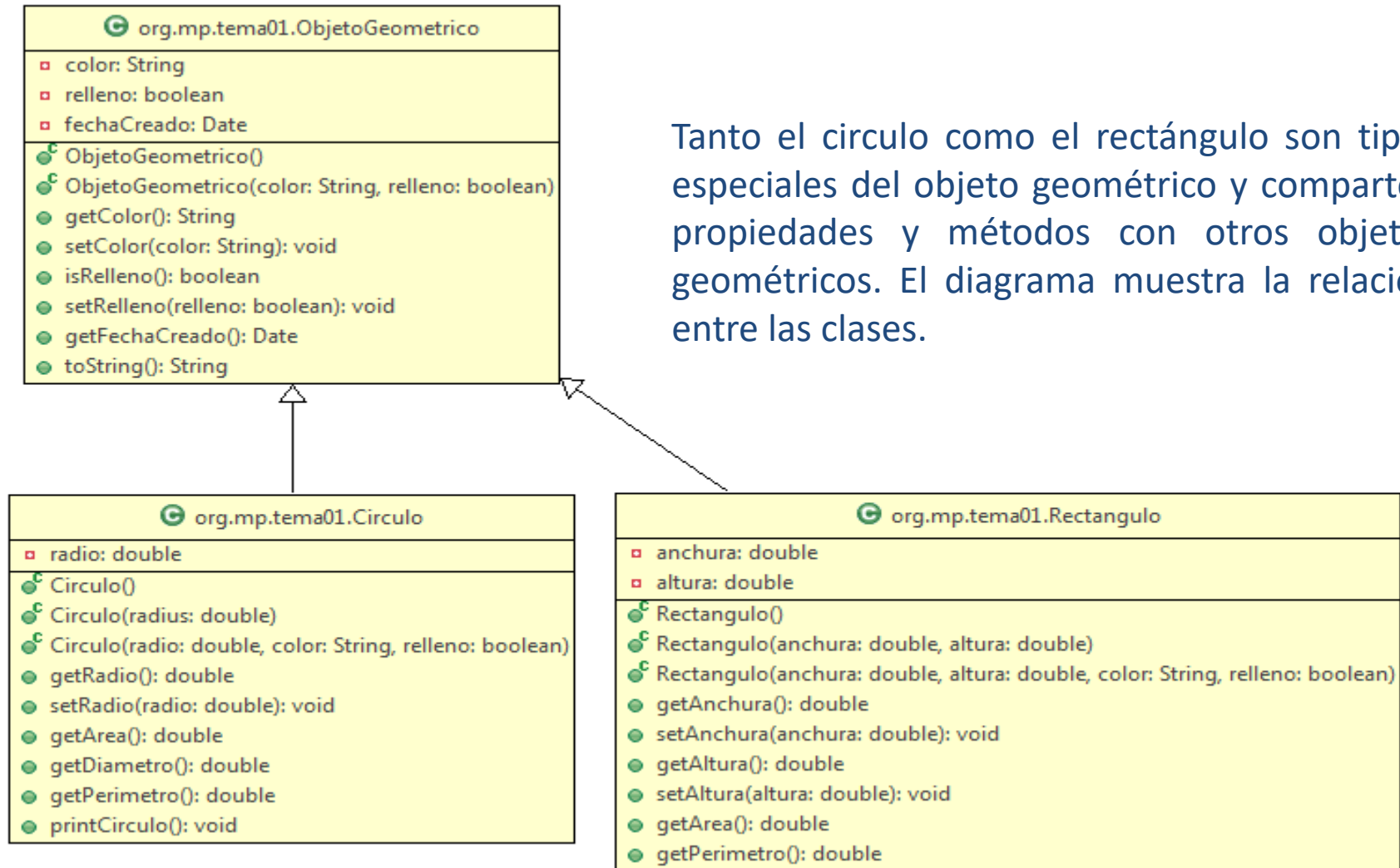
En Java una clase **C1** que hereda de otra C2 es llamada **subclase**, clase **hija** o clase **derivada**. **C2** se llama **superclase**, clase **padre** o clase **base**. La subclase hereda todas las propiedades y métodos accesibles de la superclase y puede añadir otras propiedades y métodos nuevos.

El Circulo hereda todos los propiedades y métodos accesibles y añade la propiedad **radio** y también los métodos **getArea()** y **getPerimetro()**

Sintaxis:

```
public class Subclase extends Superclase
```

```
public class Circulo extends ObjetoGeometrico
```



Tanto el circulo como el rectángulo son tipos especiales del objeto geométrico y comparten propiedades y métodos con otros objetos geométricos. El diagrama muestra la relación entre las clases.

```

1 package org.mp.tema01;
2 import java.util.Date;
3
4 public class ObjetoGeometrico {
5
6     private String color = "blanco";
7     private boolean relleno = false;
8     private Date fechaCreado;
9     /**Constructor por defecto*/
10    public ObjetoGeometrico() {
11        super();
12        fechaCreado = new Date();
13    }
14    /**Constructor especificando el color y el relleno */
15    public ObjetoGeometrico(String color, boolean relleno) {
16        super();
17        fechaCreado = new Date();
18        this.color = color;
19        this.relleno = relleno;
20    }
21    /** Devuelve el color*/
22    public String getColor() {
23        return color;
24    }
25    /** Pone un color nuevo*/
26    public void setColor(String color) {
27        this.color = color;
28    }
29    /** Devuelve el relleno*/
30    public boolean isRelleno() {
31        return relleno;
32    }
33    /** Pone un relleno nuevo*/
34    public void setRelleno(boolean relleno) {
35        this.relleno = relleno;
36    }
37    /** Devuelve la fecha en que fué creado*/
38    public Date getFechaCreado() {
39        return fechaCreado;
40    }
41    /** Devuelve un string que representa ese objeto */
42    public String toString() {
43        return "Creado el " + fechaCreado + "\n color: " + color
44            + " y relleno: " + relleno;
45    }
46 }

```

```

1 package org.mp.tema01;
2 public class Rectangulo
3     extends ObjetoGeometrico {
4     private double anchura;
5     private double altura;
6
7     public Rectangulo() {
8     }
9     public Rectangulo(
10         double anchura, double altura) {
11         this.anchura = anchura;
12         this.altura = altura;
13     }
14     public Rectangulo(
15         double anchura, double altura, String color, boolean relleno) {
16         this.anchura = anchura;
17         this.altura = altura;
18         setColor(color);
19         setRelleno(relleno);
20     }
21     /** Devuelve la anchura */
22     public double getAnchura() {
23         return anchura;
24     }
25     /** Pone una nueva anchura */
26     public void setAnchura(double anchura) {
27         this.anchura = anchura;
28     }
29     /** Devuelve la altura */
30     public double getAltura() {
31         return altura;
32     }
33     /** Pone una nueva altura */
34     public void setAltura(double altura) {
35         this.altura = altura;
36     }
37     /** Devuelve el área */
38     public double getArea() {
39         return anchura * altura;
40     }
41     /** Devuelve el perímetro */
42     public double getPerimetro() {
43         return 2 * (anchura + altura);
44     }
45 }

```

```

1 package org.mp.tema01;
2
3 public class Circulo extends ObjetoGeometrico {
4     private double radio;
5
6     public Circulo() {
7     }
8     public Circulo(double radius) {
9         this.radio = radius;
10    }
11    public Circulo(double radio, String color, boolean relleno) {
12        this.radio = radio;
13        setColor(color);
14        setRelleno(relleno);
15    }
16    /** Devuelve el radio */
17    public double getRadio() {
18        return radio;
19    }
20    /** Pone un nuevo radio */
21    public void setRadio(double radio) {
22        this.radio = radio;
23    }
24    /** Devuelve el área */
25    public double getArea() {
26        return radio * radio * Math.PI;
27    }
28    /** Devuelve el diámetro */
29    public double getDiametro() {
30        return 2 * radio;
31    }
32    /** Devuelve el perímetro */
33    public double getPerimetro() {
34        return 2 * radio * Math.PI;
35    }
36    /** Muestra información sobre el círculo */
37    public void printCirculo() {
38        System.out.println("El círculo fue creado " + getFechaCreado()
39            + " y el radio es " + radio);
40    }
41 }

```



```

1 package org.mp.tema01;
2
3
4 public class TestCirculoRectangulo {
5     public static void main(String[] args) {
6
7         Circulo circulo =
8             new Circulo(1);
9         System.out.println("Circulo " + circulo.toString());
10        System.out.println("Su color es " + circulo.getColor());
11        System.out.println("El radio es " + circulo.getRadio());
12        System.out.println("El área es " + circulo.getArea());
13        System.out.println("El diámetro es " + circulo.getDiametro());
14
15        Rectangulo rectangulo =
16            new Rectangulo(2, 4);
17        System.out.println("\nRectángulo " + rectangulo.toString());
18        System.out.println("El área es " + rectangulo.getArea());
19        System.out.println("El perímetro es " +
20            rectangulo.getPerimetro());
21    }
22 }

```



Salida

```

Circulo Creado el Mon Oct 26 17:26:38 CET 2015
color: blanco y relleno: false
Su color es blanco
El radio es 1.0
El área es 3.141592653589793
El diámetro es 2.0

```

```

Rectángulo Creado el Mon Oct 26 17:26:38 CET 2015
color: blanco y relleno: false
El área es 8.0
El perímetro es 12.0

```

Importante

Una subclase no es un subconjunto de su superclase. Una subclase normalmente contiene más información y métodos que la superclase.

Las propiedades privadas de una superclase no son accesibles fuera de la clase, por tanto no pueden ser usadas directamente en la subclase. Se pueden utilizar los métodos acceso/modificadores definidos en la superclase como públicos para acceder a ellas.

La herencia se usa para modelar relaciones es – un.

Algunos lenguajes permiten derivar una subclase de distintas clases. Esta capacidad se conoce como *herencia múltiple*. Una clase en Java hereda de una única clase, la superclase. Esto se conoce como *herencia simple*.

¿El constructor de la superclase se hereda?

No. No se hereda.

Tiene que invocarse explícita o implícitamente.

Explícitamente usando la palabra reservada **super**.

Si la palabra reservada **super** no se usa explícitamente, se hace una llamada automática al constructor por defecto (sin argumentos) de la superclase.

public A ()	es equivalente a	public A ()
{		super ();
}		}

public A (double d)	es equivalente a	public A (double d)
//algunas sentencias		super ();
}		//algunas sentencias }

Uso de la palabra reservada **super**

La palabra reservada **super** hace referencia a la superclase de la clase donde aparece y sirve para invocar métodos y constructores de la superclase.

A diferencia de propiedades y métodos, los constructores de una superclase no los hereda la subclase. En la subclase pueden ser invocados usando **super**.

```
public Circulo(double radio, String color, boolean relleno) {  
    super (color, relleno);  
    this.radio = radio  
}
```

super debe ser la primera sentencia en el constructor

Encadenar constructores

```
public class Profesor extends Empleado {
    public static void main(String [] args) {
        new Profesor( );
    }
    public Profesor ( ) {
        System.out.println(" (4) Invocado el constructor sin argumentos de Profesor");
    }
}

public class Empleado extends Persona ( ) {
    public Empleado ( ){
        this("(2) Invoca al constructor sobrecargado de Empleado");
        System.out.println(" (3) Invocado el constructor sin argumentos de Empleado");
    }
    public Empleado (String s){
        System.out.println(s);
    }
}

public class Persona {
    public Persona ( ) {
        System.out.println(" (1) Invocado el constructor sin argumentos de Persona");
    }
}
```




Salida

- (1) Invocado el constructor sin argumentos de Persona
- (2) Invoca al constructor sobrecargado de Empleado
- (3) Invocado el constructor sin argumentos de Empleado
- (4) Invocado el constructor sin argumentos de Profesor

Impacto de una superclase sin constructor por defecto

Se produce un error de compilación

```
 public class Manzana extends Fruta {  
    }  
    public class Fruta {  
        public Fruta (String nombre) {  
            System.out.println("Invocado el constructor de Fruta");  
        }  
    }  
}
```

Definir una subclase

Una subclase hereda de una superclase y puede también:

- Añadir nuevas propiedades
- Añadir nuevos métodos
- Sobre-escribir los métodos de la superclase.

Llamada a métodos de la superclase

Podemos sobre-escribir el método `printCirculo()` en la clase `Circulo` tal y como se indica:

```
public void printCircle() {  
    System.out.println("El ciruclo ha sido creado " +  
        super.getFechaCreado() + " y su radio es " + radio);  
}
```

Sobre-escribir métodos de la superclase

Una subclase hereda de una superclase. Algunas veces es necesario en la subclase modificar la implementación de un método definido en la superclase. Esto se conoce como método sobre-escrito.

```
public class Circle extends GeometricObject {  
    // Los otros métodos se han omitido  
  
    /** Sobreescribe el método toString definido en ObjetoGeometrico */  
    public String toString() {  
        return super.toString() + "\nradio es" + radio;  
    }  
}
```

Nota

Un método de instancia puede sobre-escribirse solo si es accesible. Es decir, un método `private` no puede sobre-escribirse ya que no es accesible desde una clase de fuera.

Igual que un método de instancia, un método estático se puede heredar pero no se puede sobre-escribir

Sobre-escritura vs Sobrecarga

Sobrecarga significa definir múltiples métodos con el mismo nombre pero distinta signatura. Sobre-escribir significa proporcionar una nueva implementación de un método en una subclase.

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    //Este método sobre-escribe el método de B  
    public void p(double i) {  
        System.out.println(i);  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    //Este método sobrecarga el método de B  
    public void p(int i) {  
        System.out.println(i);  
    }  
}
```

La clase Object y sus Métodos

Cualquier clase en Java es descendiente de la clase `java.lang.Object`. Cuando definimos una clase y no especificamos herencia, la superclase de esa clase es `Object`.

```
public class Circulo {  
    ...  
}
```

Equivalente

```
public class Circulo extends Object {  
    ...  
}
```

El método `toString()` en `Object`

Este método devuelve un string que representa al objeto. La implementación por defecto devuelve un string que contiene el nombre de la clase de la que el objeto es una instancia, el carácter `@` y un número que representa la dirección de memoria del objeto en hexadecimal.

```
Circulo circulo = new Circulo();
```

```
System.out.println(circulo.toString()); // mostraría Circulo@15672e5
```

Ese mensaje no es muy útil. Normalmente sobre-escribimos el método para que nos devuelva una string que represente al objeto.

Polimorfismo

Significa que una variable de una superclase puede referirse a un objeto subtipo.

Los tres pilares de la POO son la encapsulación, herencia y polimorfismo. Hemos aprendido los dos primeros, en esta sección veremos el tercero.

Primero vamos a definir dos términos muy útiles: **subtipo y supertipo**.

Una clase define un tipo. Un tipo definido por una subclase se llama **subtipo** y el tipo definido en una superclase se llama **supertipo**.

Según esto, Circulo es un subtipo de ObjetoGeometrico y ObjetoGeometrico es un supertipo de Circulo.

La herencia establece relación entre la subclase y la superclase. Una subclase es una especialización de la superclase, una instancia de la subclase es también una instancia de la superclase pero no al revés. Cualquier circulo es un objeto geométrico pero cualquier objeto geométrico no es un circulo.

```
public class DemoPolimorfismo {
    public static void main(String[] args) {
        m(new EstudianteGraduado());
        m(new Estudiante());
        m(new Persona());
        m(new Object());
    }

    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

public class EstudianteGraduado extends Estudiante{
}

public class Estudiante extends Persona {
    public String toString() {
        return "Estudiante";
    }
}

public class Persona extends Object {
    public String toString() {
        return "Persona";
    }
}
```

El método m recibe un parámetro de tipo Object. Invocamos ese método con cualquier objeto.

Un objeto de un subtipo puede usarse siempre que se requiera un valor del supertipo. Esto se conoce como **polimorfismo**.

Cuando el método m(Object x) se ejecuta, se invoca al método toString() con x.

x puede ser una instancia de:

- EstudianteGraduado
- Estudiante
- Persona
- Object

Las clases EstudianteGraduado, Estudiante, Persona y Object tienen sus implementaciones del método toString(). Qué implementación utilizar se determina dinámicamente por la JVM en tiempo de ejecución. Esto se conoce como **enlace dinámico**.

Enlace dinámico

El enlace dinámico trabaja como sigue: supongamos que un objeto o es una instancia de las clases C_1, C_2, \dots, C_{N-1} y C_N , donde C_1 es una subclase de C_2 , C_2 es una subclase de C_3 , ..., y C_{N-1} , es una subclase de C_N . Es decir, C_N es la clase más general y C_1 es la clase más específica. En Java, C_N es la clase Object. Si o invoca al método p , la JVM busca la implementación del método p en C_1, C_2, \dots, C_{N-1} y C_N en ese orden hasta que lo encuentra. Cuando ha encontrado la implementación, la búsqueda para y la primera implementación encontrada es invocada.



Dado que o es una instancia de C_1 , o es también una instancia de C_2, C_3, \dots, C_N .

Casting de Object

Hemos usado el operador casting para convertir variables de un tipo primitivo en otro. Puede usarse también para convertir un objeto de un tipo de clase en otro dentro de una jerarquía de clases. La sentencia:

```
m(new Estudiante());
```

Asigna el objeto `new Estudiante()` al parámetro de tipo `Object`. Esa sentencia es equivalente a:

```
Object o = new Estudiante(); // Casting implícito  
m(o);
```

¿Cuándo es necesario un casting?

Supongamos que tenemos una asignación del tipo:

```
Estudiante e = o;
```

Se produce un error de compilación. Un objeto de la clase `Estudiante`, `e`, es también una instancia de la clase `Object` pero una instancia de `Object` no es necesariamente una instancia de `Estudiante`. Para indicarle al compilador que `o` es un objeto de `Estudiante` se debe usar un casting explícito. La sintaxis es similar a la que usamos con tipos primitivos:

```
Estudiante e = (Estudiante) o; // Casting explícito
```

El operador instanceof

Se usa este operador para comprobar cuando un objeto es una instancia de una clase:

```
Object miObjeto = new Circle();

... // Algunas líneas de código
/** Realiza un casting si miObjeto es una instancia de Circulo*/

if (miObjeto instanceof Circulo) {
    System.out.println("El diámetro del circulo es" +
        ((Circulo)miObjeto).getDiametro());
    ...
}
```

El método de `Object` `equals`

Igual que el método `toString()`, el método `equals(Object)` es un método definido en la clase **Object**

Su signatura es: **public boolean** `equals(Object o)`

Este método comprueba cuando dos objetos son iguales. La sintaxis de llamada es :
`objeto1.equals(objeto2);`

La implementación por defecto del método en la clase **Object** es:

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

Esta implementación comprueba cuando las referencias apuntan al mismo objeto. Sobre-escribiremos este método en nuestras clases para comprobar cuando dos objetos distintos tienen el mismo contenido.

```
public boolean equals(Object o) {  
    if (o instanceof Circulo) {  
        return radio == ((Circulo)o).radio;  
    }  
    else  
        return false;  
}
```


Visibilidad de datos y métodos

ACCESIBLE DESDE

Modificador	La propia clase	El propio paquete	Las clases derivadas	Diferentes paquetes
public	✓	✓	✓	✓
protected	✓	✓	✓	
(default)	✓	✓		
private	✓			

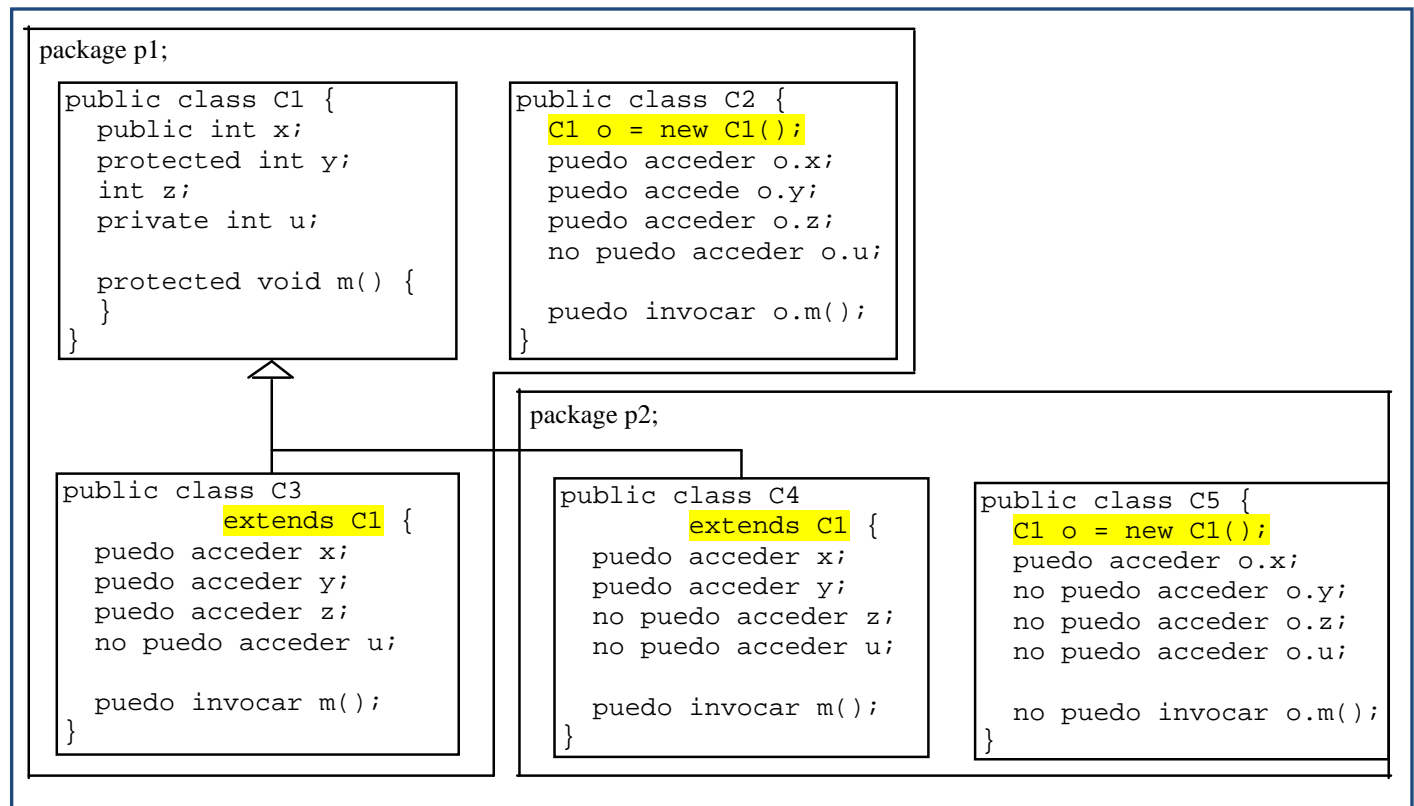
Visibilidad aumenta



private, ninguno(no se utiliza modificador), protected, public

El modificador protected

Se puede aplicar a datos y métodos en una clase. Un dato o un método protected pueden ser accesibles desde cualquier clase en el mismo paquete o en subclases aunque estas estén en diferentes paquetes.



El modificador final

- Una *clase final* no puede extenderse:

```
final class Math{  
    .....  
}
```

- Una *variable final* es una constante:

```
final static double PI = 3.141592;
```

- Un *método final* no puede sobre-escribirse en sus subclases.

- Se puede definir una nueva clase a partir de una existente. Esto se conoce como herencia. La clase nueva se llama subclase, clase hija, clase derivada o clase extendida. La clase existente se llama superclase, clase padre o clase base.
- A diferencia de las propiedades y de los métodos, el constructor de una superclase no se hereda en la subclase. Este puede invocarse solo desde el constructor de la subclase usando la palabra reservada **super**.
- Un constructor puede llamar a un constructor sobrecargado o a un constructor de la superclase. La llamada debe ser la primera sentencia en el constructor. Si no se invoca explícitamente a ningún constructor, el compilador pone **super()** como primera sentencia en el constructor lo que significa que invoca al constructor sin argumentos de la superclase.
- Para *sobre-escribir* un método, el método debe estar definido en la subclase usando la misma signatura y devolviendo el mismo tipo que en la superclase.
- Un método de instancia puede sobre-escribirse solo si es accesible. Por tanto, un método `private` no se puede sobre-escribir porque no es accesible desde una clase de fuera. Si un método definido en una subclase es `private` y también en la superclase, los dos métodos no tienen ninguna relación

- Tanto un método de instancia como un método estático pueden ser heredados. Sin embargo, un método estático no se puede sobre-escribir. Si un método estático definido en una superclase se redefine en una subclase, el método definido en la superclase es oculto.
- Todas las clases en Java son descendientes de la clase **java.lang.Object** . Si cuando se define una clase no se especifica una superclase, su superclase es **Object**.
- Si el tipo de parámetro de un método es una superclase (e.g. **Object**), podemos pasar a ese método un parámetro que sea un objeto de cualquier clase descendiente (e.g. **Circulo**) Esto es conocido como **polimorfismo**.
- Es posible castear una instancia de una subclase a una variable de una superclase porque una instancia de una subclase es *siempre* una instancia de su superclase. Cuando el casting es de una instancia de una superclase a una variable de una de sus subclases, debe hacerse una casting explícito para confirmar nuestra intención al compilador utilizando (NombreSubclase) como notación del casting.
- Una clase define un tipo. Un tipo definido en una subclase se llama *subtipo* y un tipo definido por su superclase se llama *supertipo*.

- Cuando se invoca a un método de instancia con una variable referencia, el tipo actual de la variable decide que implementación del método se usa en tiempo de ejecución. Esto se conoce como **enlace dinámico**
- Se usa **obj instanceof ClaseA** para comprobar si un objeto es una instancia de una clase.
- Se usa el modificador **protected** para evitar que los datos y los métodos sean accesibles por no subclases de paquetes diferentes.
- Se usa el modificador **final** para indicar que una clase final no se puede extender y que un método final no se puede sobre-escribir.



¡MUCHAS GRACIAS!



UNIVERSIDAD DE ALMERÍA

