



UNIVERSIDAD DE ALMERÍA

# Grado en Ingeniería Informática

## Introducción a la Programación

### 2021-2022



# Tema 3. Clases y Objetos

## Paquetes

Se usan para organizar una serie de clases relacionadas. Cada paquete consta de un conjunto de clases. Los **paquetes** son contenedores que nos permiten guardar las clases en compartimentos separados para controlar accesibilidad, seguridad y eficiencia.

Java proporciona varios paquetes predefinidos:

**java.io**: Contiene clases necesarias para entrada y salida.

**java.lang**: Contiene clases esenciales, `String`, `StringBuffer`, etc. Se importan implícitamente sin necesidad de la sentencia **`import`**.

**java.util**: contiene clases para el manejo de estructuras de datos, fechas, números aleatorios, entrada de datos por teclado (`Scanner`)

**java.applet**: Contiene clases necesarias para crear applets, es decir, programas que se ejecutan en la ventana del navegador.

**java.awt**: Contiene clases para crear aplicaciones GUI (Interfaces Gráficas de Usuarios).

**java.net**: Se usa en combinación con las clases del paquete `java.io` para leer y escribir datos en la red.

# Tema 3. Clases y Objetos

## Uso de paquetes en un programa. Sentencia **import**

Para utilizar en un programa una clase de un paquete tenemos que poner:

```
<nombre del paquete> . <nombre de la clase>;
```

Por ejemplo:

```
java.awt.image.ColorModel
```

Esta notación se denomina punteada o con punto y resulta bastante tedioso. Para evitar esto, se utiliza la sentencia **import** al principio del programa:

```
import <nombre del paquete> . <nombre de la clase>;
```

O bien

```
import <nombre del paquete> . *;
```

y así importamos todas las clases de ese paquete.

Cualquier referencia a lo largo del programa de una clase de ese paquete, evitará la notación punteada.

# Tema 3. Clases y Objetos

## Modificadores de acceso

Los modificadores de acceso pueden ser para clases y para métodos. Una clase será **visible** por otra dependiendo de si se ubican en el mismo paquete y de los modificadores de acceso que utilice (estos alteran la visibilidad, permitiendo que se muestre u oculte).

**Modificadores de acceso para clases.** Visibilidad por defecto (sin modificador de acceso), una clase es visible por las clases del mismo paquete. Visibilidad total (**public**), una clase es visible para todas las clases externas al paquete y para las del propio paquete. Para que un miembro de una clase sea visible ésta debe serlo.

**Modificadores de acceso para miembros (atributos y métodos).** Visibilidad por defecto (sin modificador), un miembro es visible desde las clases del mismo paquete, pero invisible desde clases externas a él. Modificador de acceso **public** y **private**. Miembro **private** invisible desde fuera de la clase. Miembro **public** otorga visibilidad total a clases del mismo paquete y clases externas a él.

# Tema 3. Clases y Objetos

## Uso de la clase **Scanner** para entrada de datos por teclado

Java utiliza **System.out** para referirse al dispositivo estándar de salida y **System.in** para el dispositivo estándar de entrada, normalmente la pantalla (consola) y el teclado respectivamente.

La entrada por teclado no está directamente soportada en Java, pero podemos usar la clase **Scanner** para crear un objeto que lea una entrada de **System.in**, tal y como se expresa a continuación:

```
Scanner entrada = new Scanner(System.in);
```

Esta clase **Scanner** está en el paquete **java.util**, por lo tanto, sería necesario importarlo.

Esto crearía un objeto de la clase **Scanner** y podríamos utilizar métodos de dicha clase para leer distintos tipos de datos.

# Tema 3. Clases y Objetos

Algunos métodos de la clase **Scanner**

Método	Descripción	Uso
<code>nextInt()</code>	Lee un número como tipo int	<code>int valor = entrada.nextInt();</code>
<code>nextLong()</code>	Lee un número como tipo long	<code>long valor = entrada.nextLong();</code>
<code>nextDouble()</code>	Lee un número como tipo double	<code>double valor = entrada.nextDouble();</code>
<code>nextFloat()</code>	Lee un número como tipo float	<code>float valor = entrada.nextFloat();</code>
<code>next()</code>	Lee una cadena	<code>String cadena = entrada.next();</code>
<code>nextLine()</code>	Lee una línea de texto	<code>String linea = entrada.nextLine();</code>

Previamente el objeto `entrada` deberá estar instanciado y creado.

## Ejemplo de programa con entrada por teclado

```
1 package org.ip.tema03;
2
3 import java.util.Scanner;
4
5 public class EuclidesInteractivo {
6
7     private static Scanner entrada;
8
9     public static int mcdEuclides(int dato1, int dato2) {
10         int aux;
11         while (dato1 % dato2 != 0) {
12             aux = dato1;
13             dato1 = dato2;
14             dato2 = aux % dato1;
15         }
16         return dato2;
17     }
18
19     public static void main(String[] args) {
20         entrada = new Scanner(System.in);
21
22         System.out.println("Introduzca el primer valor");
23         int dato1 = entrada.nextInt();
24         System.out.println("Introduzca el segundo valor");
25         int dato2 = entrada.nextInt();
26         System.out.println("El MCD de " + dato1 + " y " + dato2 + " es "
27             + mcdEuclides(dato1, dato2));
28     }
29 }
```

## Salida

```
Introduzca el primer valor
7
Introduzca el segundo valor
9
El MCD de 7 y 9 es 1
```

# Tema 3. Clases y Objetos

## Representación de clases

Para representar clases se suele utilizar una notación gráfica que simplifica bastante la descripción de la clase. La más estándar se denomina **UML** (Lenguaje Unificado de Modelado).

Una clase se representa con un rectángulo dividido en tres partes:

- ✓ El **nombre de la clase**

(identifica la clase de forma unívoca)

- ✓ Sus **atributos**

(datos asociados a los objetos de la clase)

- ✓ Sus **operaciones o métodos**

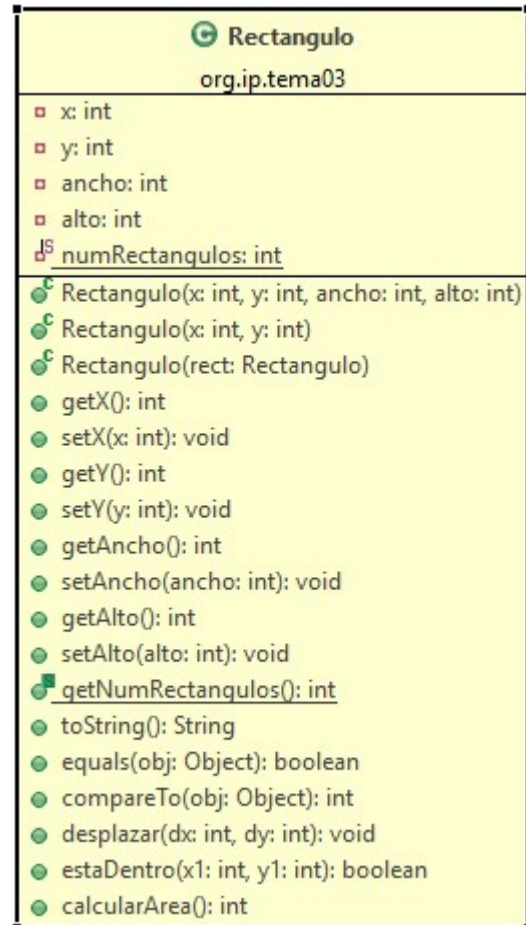
(comportamiento de los objetos de esa clase)

En **Eclipse** podemos generar estos diagramas con diferentes plug-in, uno de ellos es **Amateras**.



# Tema 3. Clases y Objetos

## Ejemplo de diagrama de clase con Amateras



# Tema 3. Clases y Objetos



## Paso de objetos a métodos

Podemos pasar objetos a métodos como parámetros. El siguiente programa pasa el objeto **miCirculo** como parámetro al método **printCirculo**.

```

1 package org.ip.tema03;
2
3 public class TestCirculo1 {
4
5     public static void printCirculo(Circulo1 c) {
6         System.out.printf("El area del circulo con radio %4.1f es %5.2f",
7             c.getRadio(), c.getArea());
8     }
9
10    public static void main(String[] args) {
11        Circulo1 miCirculo = new Circulo1(5.0);
12        printCirculo(miCirculo);
13    }
14 }

```

Circulo1 org.ip.tema03	
radio: double	
<u>numCirculos: int</u>	
Circulo1(radio: double)	
getRadio(): double	
setRadio(radio: double): void	
<u>getNumCirculos(): int</u>	
getArea(): double	

Salida

El area del circulo con radio 5,0 es 78,54



## Paso de objetos a métodos

Java utiliza un modo de paso de parámetros: *paso por valor*. En el código anterior, el valor de **miCirculo** se pasa al método **printCirculo**. El valor es una referencia a un objeto de la clase **Circulo1**. A continuación, vamos a ver la diferencia entre pasar un valor de tipo primitivo y pasar un valor de una referencia a un objeto.

```

1 package org.ip.tema03;
2
3 public class PasaObjeto {
4
5     public static void printAreas(Circulo1 c, int n) {
6         System.out.println("Radio \t\tArea");
7         while (n >= 1) {
8             System.out.println(c.getRadio() + "\t\t" + c.getArea());
9             c.setRadio(c.getRadio() + 1);
10            n--;
11        }
12    }
13
14    public static void main(String[] args) {
15        Circulo1 miCirculo = new Circulo1(1);
16        int n = 5;
17        printAreas(miCirculo, n);
18        System.out.println("El radio es " + miCirculo.getRadio());
19        System.out.println("n es " + n);
20    }
21 }

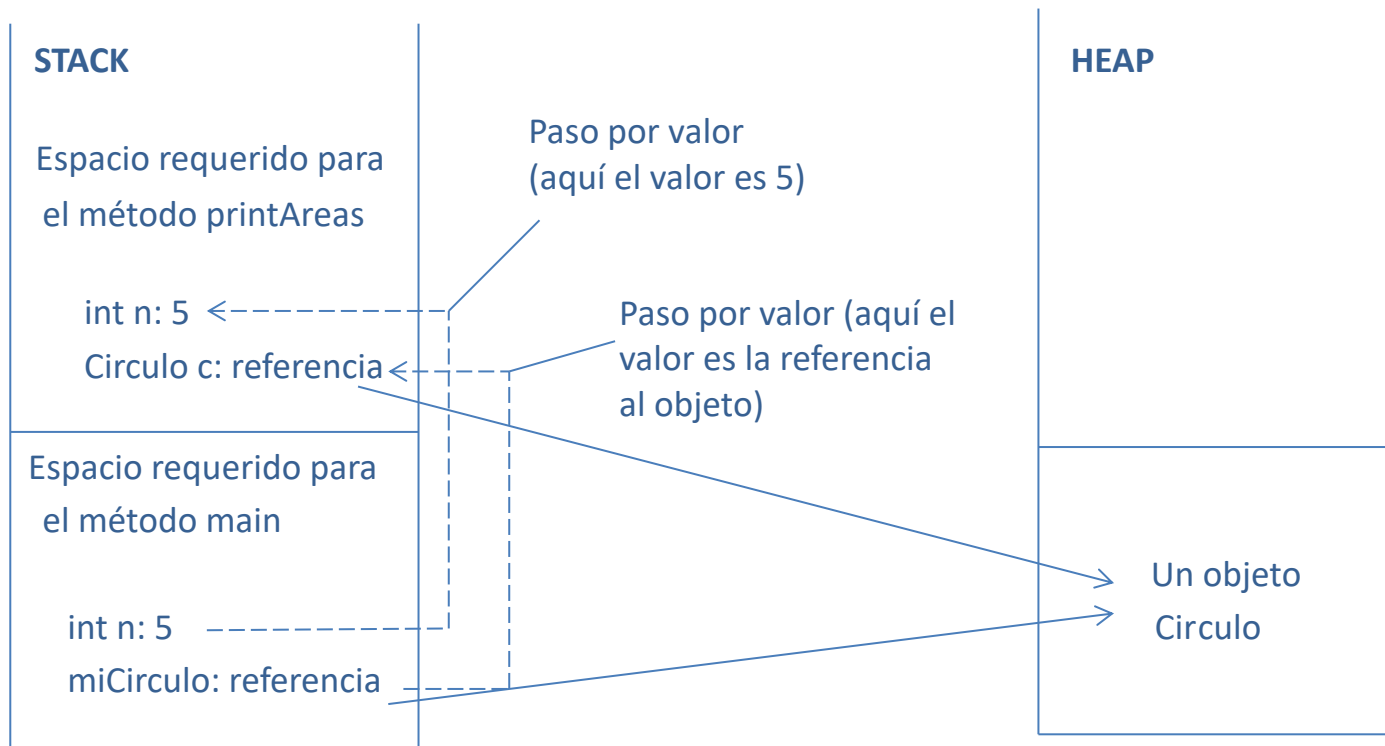
```

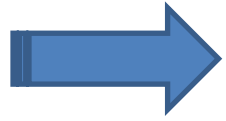
Salida

Radio	Area
1.0	3.141592653589793
2.0	12.566370614359172
3.0	28.274333882308138
4.0	50.26548245743669
5.0	78.53981633974483
El radio es 6.0	
n es 5	

## ➡ Estado de la memoria

Cuando se hace una llamada a un método, el sistema operativo almacena los parámetros y las variables en una zona de memoria conocida como *stack* (*pila*) y los objetos en la zona conocida como *heap*.





## Paso de objetos a métodos

**Referencia.** Antes de seguir hay que comprender el concepto de *dirección de memoria*. La memoria de un ordenador está formada por pequeños **bloques de memoria** consecutivos de un tamaño determinado que dependerá del tipo de hardware. Cada uno de esos bloques se identifica mediante un número único, que se denomina *dirección de memoria* (se numera en hexadecimal).

Cualquier dato almacenado en memoria ocupará una serie de bloques consecutivos y puede ser identificado mediante la dirección de memoria del *primer bloque* que ocupa. A esa primera dirección de memoria que identifica a un dato se le denomina en Java ... **referencia**.

Cualquier dato en memoria, incluidos los objetos, se identifica mediante su **referencia**, es decir, mediante la dirección del primer bloque que ocupa en la memoria.



## Paso de objetos a métodos

**Variables referencia.** Antes de construir objetos necesitamos declarar variables cuyo tipo sea una clase. La declaración sigue las mismas reglas que las variables de tipos primitivos.

`<nombre de clase> <nombre de variable>;`

La diferencia entre una variable de tipo primitivo y una variable de tipo referencia es que, mientras una variable de tipo primitivo almacena directamente un valor, una variable de tipo clase almacena la **referencia** de un objeto.

**Referencia null.** El valor literal **null** es una referencia nula. Dicho de otra forma, es una referencia a *ningún* bloque de memoria. Cuando declaramos una variable referencia se inicializa por defecto a **null**. Cuidado al intentar acceder a los miembros de una referencia nula (null), ya que produce un error, en algunos casos del tipo *Null pointer exception*



## Paso de objetos a métodos

Recordemos que cuando se pasa como parámetro un tipo de **dato primitivo**, el *parámetro real* pasa **el valor** al *parámetro formal* (se copia) y si éste se modifica en el método, el *parámetro real* no se ve afectado.

**MUY IMPORTANTE:** Cuando se pasa como *parámetro un tipo referencia*, se pasa la **referencia al objeto**. En este caso, **c** (*parámetro formal*) contiene la referencia al objeto que también es referenciado por **miCirculo** (*parámetro real*). Por ello, un cambio en las propiedades del objeto **c** en el método **printAreas** tiene el mismo efecto en la variable **miCirculo** declarada y creada en el método **main**. El paso **por valor con referencias** se puede describir como un **paso compartido**, es decir, el objeto referenciado en el método (*parámetro formal*) es el mismo objeto que el pasado como *parámetro real* (objeto externo al método).



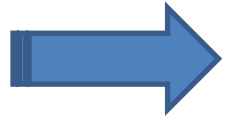


## Paso de objetos a métodos (Aclaración)

El término *pasar por valor* y su significado dependen de cómo se perciba el funcionamiento del programa. El significado general es que se logra una copia de sea lo que sea lo que se pasa, pero la pregunta real es cómo se piensa en lo que se pasa. Cuando se *pasa por valor*, hay dos visiones claramente distintas:

**Java pasa todo por valor.** Cuando se pasan *datos primitivos* a un método, se logra una copia aparte del dato. Cuando se pasa *una referencia a un método*, se obtiene una referencia al método, se puede lograr una copia de la referencia. Todo se pasa por valor. Por supuesto, se supone que siempre se piensa (y se tiene cuidado en qué) que se están pasando referencias, pero parece que el diseño de Java ha ido mucho más allá, permitiéndote ignorar (la mayoría de las veces) que se está trabajando con una referencia. Es decir, parece permitirnos pensar en la referencia como si se tratara *del objeto* puesto que implícitamente se *desreferencia* cuando se hace una llamada a un método.



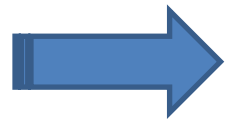


## Paso de objetos a métodos (Aclaración)

Cuando se *pasa por valor*, hay dos visiones claramente distintas (continuación):

**Java pasa los tipos de datos primitivos por valor** (sin que haya parámetros), **pero los objetos se pasan por referencia**. Ésta es la visión de que la referencia es un alias del objeto, por lo que no se piensa en el paso de referencias, sino que se dice *estoy pasando el objeto*. Dado que no se logra una copia local del objeto, cuando se pasa a un método, por lo que claramente, los objetos no se pasan por valor, sino por referencia.

Habiendo visto ambas perspectivas, y tras decir que *depende de cómo vea cada uno lo que es una referencia*. Al final, no es tan importante, lo que es importante es que se entienda que pasar una referencia permite que el objeto que hizo la llamada pueda cambiar de forma inesperada.



## Paso de objetos a métodos (Ejemplo)

Ejemplo de paso de objetos a métodos:

```

1 package org.ip.tema03;
2
3 public class Persona {
4
5     private String nombre;
6
7     public String getNombre() {
8         return nombre;
9     }
10
11     public void setNombre(String nombre) {
12         this.nombre = nombre;
13     }
14 }

```

```

1 package org.ip.tema03;
2
3 public class PasaObjetosMetodos {
4
5     private static void cambiarNombre(Persona p) {
6         //p = new Persona(); // nuevo valor al parametro
7         p.setNombre("Antonio"); // aqui tendra otro valor
8         System.out.println("El nombre en el metodo es: " + p.getNombre());
9     }
10
11     public static void main(String[] args) {
12         Persona p = new Persona(); // creamos un objeto
13         p.setNombre("Juan"); // le asignamos un valor
14         System.out.println("El nombre antes del metodo es: " + p.getNombre());
15         cambiarNombre(p); // llamamos a un metodo para que cambie de valor
16         System.out.println("El nombre despues del metodo es: " + p.getNombre());
17     }
18 }

```

# Tema 3. Clases y Objetos

## Composición

En Java existen dos formas de **reutilizar** código: la **composición** y la **herencia**.

La **composición** consiste en definir clases cuyos atributos son **objetos**.

Supongamos que tenemos la clase **Punto**:

Punto org.ip.tema03
<ul style="list-style-type: none"> <li>▣ x: int</li> <li>▣ y: int</li> </ul>
<ul style="list-style-type: none"> <li>● Punto(x: int, y: int)</li> <li>● toString(): String</li> <li>● equals(obj: Object): boolean</li> <li>● getX(): int</li> <li>● setX(x: int): void</li> <li>● getY(): int</li> <li>● setY(y: int): void</li> <li>● desplazar(dx: int, dy: int): void</li> </ul>

```

1 package org.ip.tema03;
2 public class Punto {
3     private int x;
4     private int y;
5     public Punto(int x, int y) {
6         super();
7         this.x = x;
8         this.y = y;
9     }
10    @Override
11    public String toString() {
12        return "Punto [x=" + x + ", y=" + y + "]";
13    }
14    @Override
15    public boolean equals(Object obj) {
16        Punto otro = (Punto) obj;
17        return x == otro.x && y == otro.y;
18    }
19    public int getX() {
20        return x;
21    }
22    public void setX(int x) {
23        this.x = x;
24    }
25    public int getY() {
26        return y;
27    }
28    public void setY(int y) {
29        this.y = y;
30    }
31    public void desplazar(int dx, int dy) {
32        x += dx; y += dy;
33    }
34 }

```

# Tema 3. Clases y Objetos

## Composición

La clase **RectanguloCom** podemos diseñarla utilizando la clase **Punto**.

```

1 package org.ip.tema03;
2 public class RectanguloCom {
3     private Punto origen;
4     private int ancho;
5     private int alto;
6     public RectanguloCom(Punto origen, int ancho, int alto) {
7         super();
8         this.origen = origen;
9         this.ancho = ancho;
10        this.alto = alto;
11    }
12    @Override
13    public String toString() {
14        return "Rectangulo [origen=" + origen + ", ancho=" + ancho
15            + ", alto=" + alto + "]";
16    }
17    public Punto getOrigen() { return origen; }
18    public void setOrigen(Punto origen) { this.origen = origen; }
19    public int getAncho() { return ancho; }
20    public void setAncho(int ancho) { this.ancho = ancho; }
21    public int getAlto() { return alto; }
22    public void setAlto(int alto) { this.alto = alto; }
23    @Override
24    public boolean equals(Object obj) {
25        RectanguloCom otro = (RectanguloCom) obj;
26        return ancho == otro.ancho && alto == otro.alto && origen.equals(otro.origen);
27    }
28    public void desplazar(int dx, int dy) {
29        origen.desplazar(dx, dy);
30    }
31 }

```

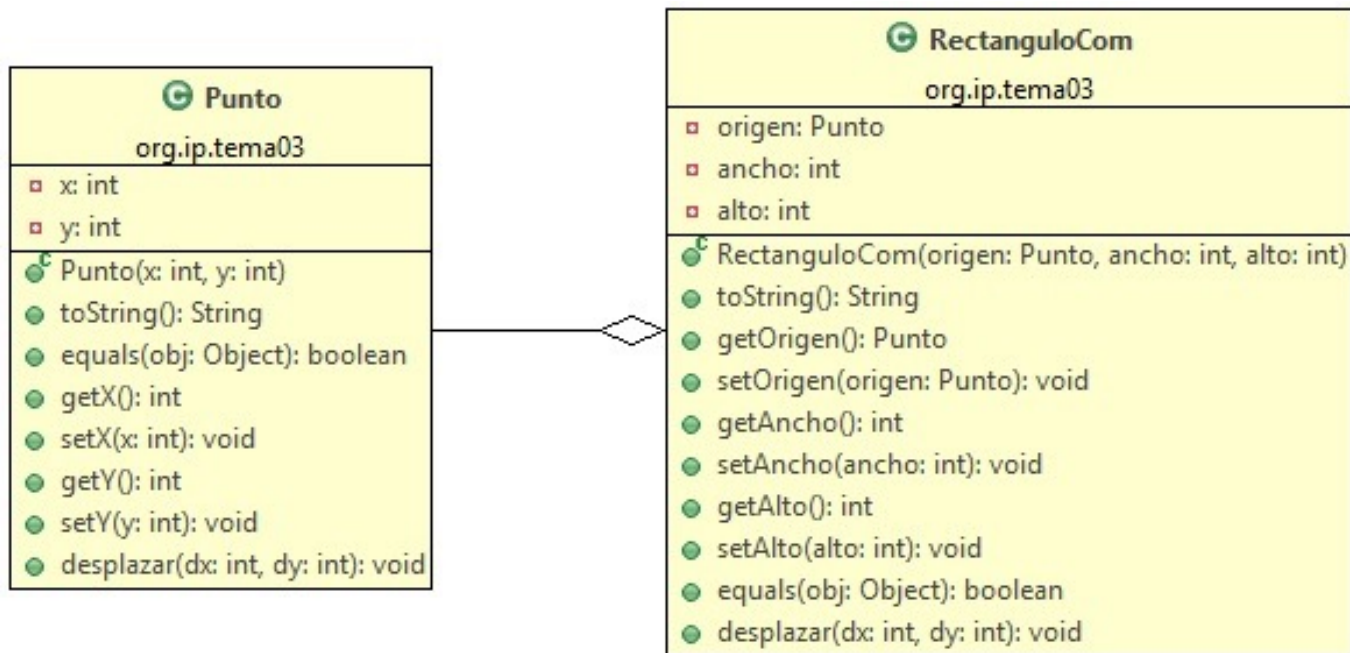
<b>RectanguloCom</b> org.ip.tema03
<ul style="list-style-type: none"> <li>origen: Punto</li> <li>ancho: int</li> <li>alto: int</li> </ul>
<ul style="list-style-type: none"> <li> RectanguloCom(origen: Punto, ancho: int, alto: int)</li> <li> toString(): String</li> <li> getOrigen(): Punto</li> <li> setOrigen(origen: Punto): void</li> <li> getAncho(): int</li> <li> setAncho(ancho: int): void</li> <li> getAlto(): int</li> <li> setAlto(alto: int): void</li> <li> equals(obj: Object): boolean</li> <li> desplazar(dx: int, dy: int): void</li> </ul>

# Tema 3. Clases y Objetos



## Composición

Diagrama de clases donde se refleja la composición



La composición crea relaciones *tiene* entre clases. Es decir, un objeto de la clase **RectanguloCom** *tiene* un objeto de la clase **Punto**.



# Tema 3. Clases y Objetos



## Composición

### Ejemplo de uso

```
1 package org.ip.tema03;
2
3 public class RectanguloComPrincipal {
4
5     public static void main(String[] args) {
6         Punto p = new Punto(5 ,10);
7         System.out.println(p.toString());
8         RectanguloCom rect1 = new RectanguloCom(p, 5, 20);
9         System.out.println(rect1.toString());
10        RectanguloCom rect2 = new RectanguloCom(new Punto(4, 4), 10, 20);
11        System.out.println(rect2.toString());
12        rect1.desplazar(2, 2);
13        System.out.println(rect1.toString());
14    }
15 }
```

### Salida

```
Punto [x=5, y=10]
Rectangulo [origen=Punto [x=5, y=10], ancho=5, alto=20]
Rectangulo [origen=Punto [x=4, y=4], ancho=10, alto=20]
Rectangulo [origen=Punto [x=7, y=12], ancho=5, alto=20]
```

# Tema 3. Clases y Objetos

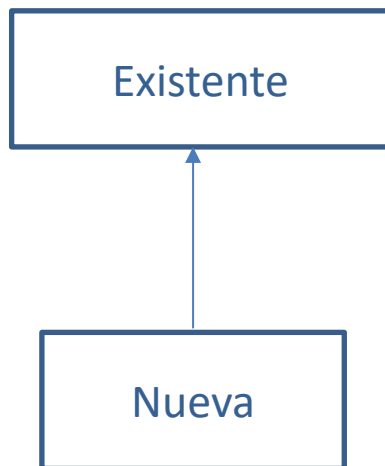
## ➡ Herencia

La **herencia** es la otra forma que existe en Java de reutilizar código.

Establece una relación entre clases del tipo *es-un*.

A partir de una clase denominada *base*, *superclase* o *padre* se crea otra denominada *clase derivada*, *subclase* o *hija*.

La relación de herencia se establece entre una clase **Nueva** y una clase **Existente**.



**Nueva** hereda todas las características de **Existente**.

**Nueva** puede definir características adicionales.

**Nueva** puede redefinir métodos heredados de **Existente**.

El proceso de herencia no afecta de ninguna forma a la superclase o clase **Existente**.

# Tema 3. Clases y Objetos

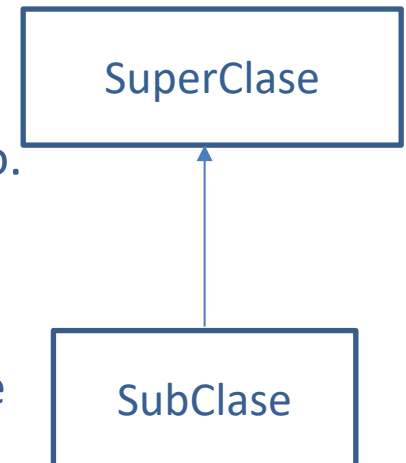
## Herencia

Una *SubClase* dispone de los mismos miembros heredados de la *SuperClase* y, habitualmente, se amplía añadiéndole nuevos atributos y métodos, aumentando su funcionalidad a la vez que se evita la repetición de código (reutilización)

La forma de expresar cuál es la *SuperClase* de la que heredamos es mediante la palabra reservada **extends**

```
class SubClase extends SuperClase { ... }
```

**Redefinición de miembros heredados.** Cuando una clase hereda una serie de miembros, en alguna ocasión puede ocurrir que interese modificar el tipo de algún atributo o redefinir un método. Este mecanismo se conoce como *Ocultación* cuando es un atributo y *Sustitución* u *Overriding* para un método. Consiste en declarar un miembro con igual nombre que uno heredado, lo que hace que éste sea ocultado o sustituido por el nuevo.





# Tema 3. Clases y Objetos



## Herencia

### Declaración de una clase derivada

```
<modificadores> class <nombre clase derivada> extends <nombre clase base> {  
    // atributos, métodos, ...  
}
```

**Todas** las clases en Java **heredan** de la clase **Object** (SuperClase por excelencia), es la raíz de la jerarquía de herencia. Los métodos **equals** y **toString** son de la clase **Object** que pueden ser redefinidos en cada SubClase.

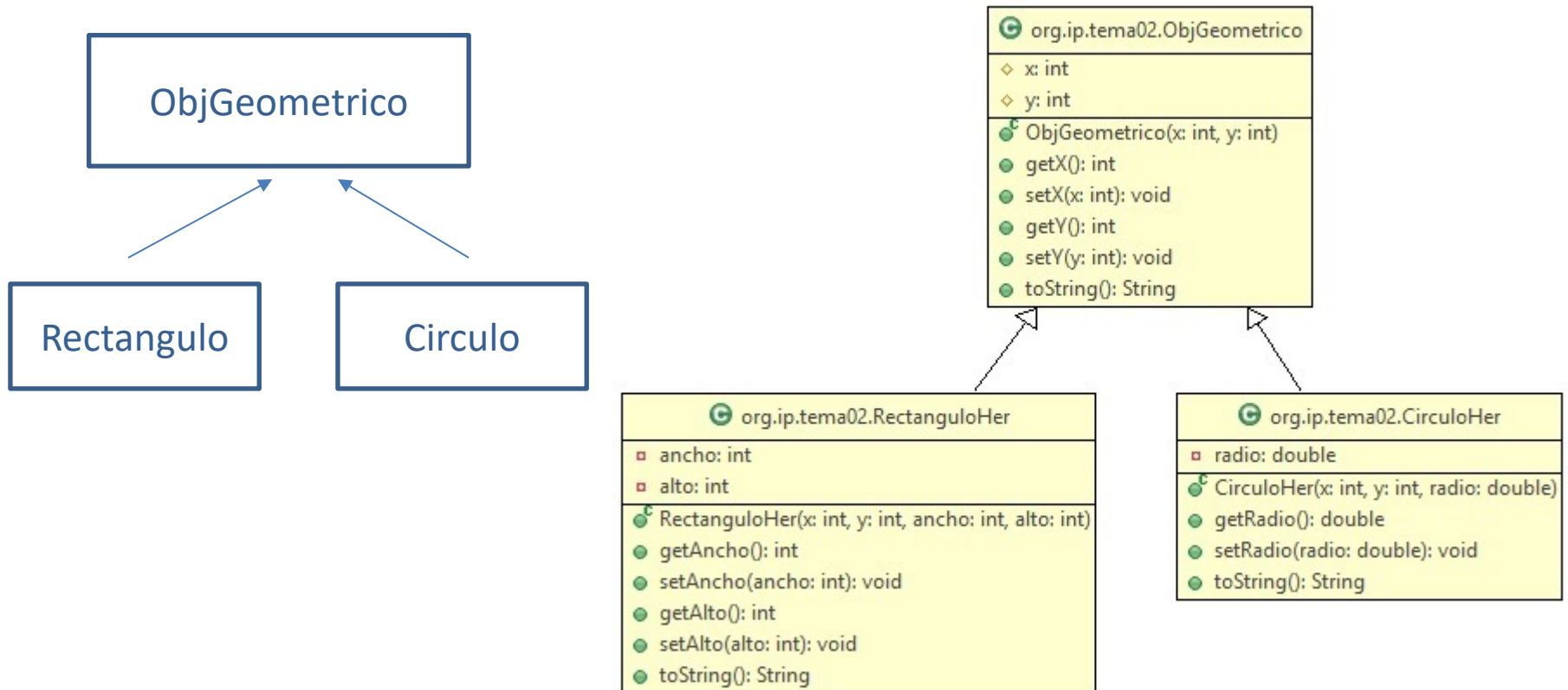
El método **toString()** devuelve una cadena que representa al objeto que se llama. Normalmente se suele realizar un *overriding* de este método en la SubClase.

El método **equals()** compara dos objetos y decide si son iguales (**true**) y **false** en caso contrario. Recordemos que el operador **==** es válido para comparar tipos primitivos, pero no sirve para comparar objetos ya que sólo examina sus referencias sin fijarse en su contenido. Para comparar contenido de objetos ... **equals()**

# Tema 3. Clases y Objetos


## Herencia

Supongamos que existe la clase **ObjGeometrico** a partir de ella podemos crear las clases **Rectangulo** y **Circulo** derivadas.



# Tema 3. Clases y Objetos

## Herencia

 org.ip.tema02.ObjGeometrico
◇ x: int
◇ y: int
● ObjGeometrico(x: int, y: int)
● getX(): int
● setX(x: int): void
● getY(): int
● setY(y: int): void
● toString(): String

```

1 package org.ip.tema02;
2
3 public class ObjGeometrico {
4     protected int x;
5     protected int y;
6     public ObjGeometrico(int x, int y) {
7         super();
8         this.x = x;
9         this.y = y;
10    }
11    public int getX() {
12        return x;
13    }
14    public void setX(int x) {
15        this.x = x;
16    }
17    public int getY() {
18        return y;
19    }
20    public void setY(int y) {
21        this.y = y;
22    }
23    @Override
24    public String toString() {
25        return "(" + x + ", " + y + ")";
26    }
27 }

```

# Tema 3. Clases y Objetos

## Visibilidad de Datos y Métodos para la Herencia

ACCESIBLE/VISIBLE DESDE

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

Visibilidad aumenta

—————→  
**private**, ninguno (no se utiliza modificador), **protected**, **public**

# Tema 3. Clases y Objetos

## Herencia

### Modificadores de acceso para la herencia

Con la aparición de la herencia podemos plantearnos algunas cuestiones, ¿se heredan todos los miembros de una clase?, si no es así, ¿cuáles son los miembros que realmente se heredan?. La respuesta es que se heredan todos, aunque los **private** no son visibles a la SubClase. No obstante, se puede acceder a ellos con un método no privado.

El modificador **protected** está pensado para la facilitar la herencia, funciona de forma similar a la visibilidad por defecto (visibilidad entre clases dentro del mismo paquete), con la diferencia de que los miembros protegidos (**protected**) serán siempre visibles para las clases que hereden, indistintamente de si la SuperClase y las SubClase están en el mismo paquete o no. Es decir, un miembro **protected** es visible en las clases del mismo paquete, pero no es visible para las clases externas, pero siempre es visible, indistintamente del paquete, desde una clase hija.

# Tema 3. Clases y Objetos

## Herencia

org.ip.tema02.RectanguloHer
<ul style="list-style-type: none"> <li>ancho: int</li> <li>alto: int</li> </ul>
<ul style="list-style-type: none"> <li>RectanguloHer(x: int, y: int, ancho: int, alto: int)</li> <li>getAncho(): int</li> <li>setAncho(ancho: int): void</li> <li>getAlto(): int</li> <li>setAlto(alto: int): void</li> <li>toString(): String</li> </ul>

**@Override** sustituye un método de la SuperClase

**super** hace referencia a la SuperClase de la clase donde se utiliza

**super()** invoca un constructor de la SuperClase y debe ser la primera instrucción de un constructor

```

1 package org.ip.tema02;
2
3 public class RectanguloHer extends ObjGeometrico {
4     private int ancho;
5     private int alto;
6
7     public RectanguloHer(int x, int y, int ancho, int alto) {
8         super(x, y);
9         this.ancho = ancho;
10        this.alto = alto;
11    }
12    public int getAncho() {
13        return ancho;
14    }
15    public void setAncho(int ancho) {
16        this.ancho = ancho;
17    }
18    public int getAlto() {
19        return alto;
20    }
21    public void setAlto(int alto) {
22        this.alto = alto;
23    }
24    @Override
25    public String toString() {
26        return "Origen del rectangulo = " + super.toString()
27            + ", ancho = " + ancho + ", alto = " + alto;
28    }
29 }

```

*super(...)* (arrow pointing to line 8)

*super* (arrow pointing to line 26)

# Tema 3. Clases y Objetos

## Herencia

org.ip.tema02.CirculoHer
radio: double
CirculoHer(x: int, y: int, radio: double)
getRadio(): double
setRadio(radio: double): void
toString(): String

Recuerde que **this** es la palabra reservada que se utiliza para indicar ... la propia clase

```

1 package org.ip.tema02;
2
3 public class CirculoHer extends ObjGeometrico {
4     private double radio;
5
6     public CirculoHer(int x, int y, double radio) {
7         super(x, y);
8         this.radio = radio;
9     }
10    public double getRadio() {
11        return radio;
12    }
13    public void setRadio(double radio) {
14        this.radio = radio;
15    }
16    @Override
17    public String toString() {
18        return "Origen del circulo = " + super.toString()
19            + ", radio = " + radio;
20    }
21 }

```



# Tema 3. Clases y Objetos



## Herencia

### Ejemplo de uso

```
1 package org.ip.tema02;
2
3 public class ObjPrincipal {
4
5     public static void main(String[] args) {
6         ObjGeometrico obj = new ObjGeometrico(1, 7);
7         System.out.println(obj.toString());
8         RectanguloHer rectangulo = new RectanguloHer(1, 4, 5, 5);
9         System.out.println(rectangulo.toString());
10        CirculoHer circulo = new CirculoHer(3, 4, 5.7);
11        System.out.println(circulo.toString());
12        System.out.println("El valor de Y en el rectangulo es "
13            + rectangulo.getY());
14    }
15 }
```

Salida

```
(1, 7)
Origen del rectangulo = (1, 4), ancho = 5, alto = 5
Origen del circulo = (3, 4), radio = 5.7
El valor de Y en el rectangulo es 4
```





**¡MUCHAS GRACIAS!**



UNIVERSIDAD DE ALMERÍA

