



UNIVERSIDAD DE ALMERÍA

Grado en Ingeniería Informática

Introducción a la Programación

2016-2017



Tema 2. Clases y Objetos

- Introducción
- Clases y objetos
- Métodos básicos
- Paquetes
- Representación de clases
- Paso de objetos a métodos
- Composición
- Herencia
- Clases Abstractas e Interfaces
- Excepciones

Tema 2. Clases y Objetos

Introducción

Principios de la orientación a objetos

Todos los programas de computadora constan de dos elementos: **código y datos**.

Sin embargo, un programa se puede organizar conceptualmente alrededor de su código o de sus datos.

En los lenguajes procedimentales, orientados a procesos o **estructurados** funcionan como “**un código que actúa sobre datos**”.

El paradigma orientado a objetos organiza el programa alrededor de los datos (es decir, objetos) y se puede caracterizar por el **control de los datos sobre el código**.

Tema 2. Clases y Objetos

Introducción

La idea principal de la **POO** es un conjunto de objetos que interactúan entre sí y que están organizados en clases. Sus principios fundamentales son:

- ❑ **Abstracción:** Consiste en olvidarnos de los detalles, constituyendo así un mecanismo fundamental para permitir la comprensión de sistemas complejos.

- ❑ **Encapsulamiento u ocultación de la información:** Consiste en la combinación de los datos y las operaciones que se pueden ejecutar con esos datos, impidiendo usos indebidos, ya que el acceso a los datos se hará a través de las operaciones que tengamos definidas (métodos.) La base del encapsulamiento o de la ocultación de la información en Java es la clase.

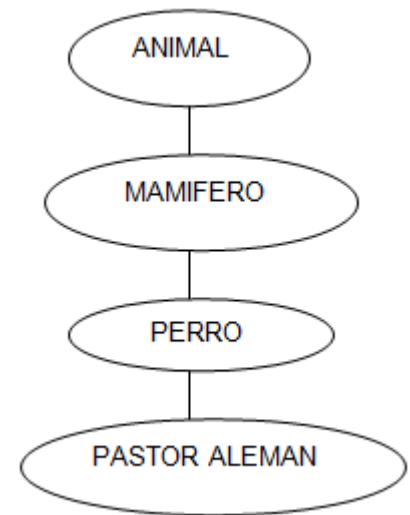
Tema 2. Clases y Objetos

Introducción

❑ **Herencia:** Consiste en la capacidad para crear nuevas clases que llamaremos descendientes o derivadas y que se construyen sobre otras ya existentes, que denominamos antecesoras, superclase o clase base, permitiendo que estas últimas (superclase) le transmitan sus propiedades a las derivadas.

Un pastor alemán tiene las características de los perros que a su vez tiene la de los mamíferos (o es un mamífero) y a su vez estos las de los animales.

Se puede establecer una jerarquía:



Tema 2. Clases y Objetos

Introducción

☐ **Polimorfismo:** Consiste en que un mismo mensaje pueda actuar sobre diferentes tipos de objetos y comportarse de modo distinto.

Tema 2. Clases y Objetos

Clases y objetos

Los dos conceptos más importantes de la programación orientada a objetos son los de **clase** y **objeto**.

Un **objeto** en su acepción o significado más amplio es cualquier cosa tanto tangible como intangible que podamos imaginar. En un programa escrito en el estilo orientado a objetos tendremos una serie de objetos interaccionando entre sí.

Para que una computadora sea capaz de crear un objeto es necesario proporcionarle una definición (también llamada plantilla o molde) y eso es lo que denominamos **clase**. Una clase es una plantilla implementada en software que describe un conjunto de objetos con atributos y comportamiento similares.

Una **instancia u objeto** de una clase es una representación concreta y específica de una clase y que reside en la memoria del ordenador.

Tema 2. Clases y Objetos

Clases y objetos

☐ Atributos

Los atributos son las **características individuales** que diferencian un objeto de otro y determinan su apariencia, estado u otras cualidades. Los atributos se guardan en **variables** denominadas **de instancia**, y cada objeto particular puede tener valores distintos para estas variables.

Las variables de instancia también denominados **miembros dato**, son declaradas en la clase pero sus valores son fijados y cambiados en el objeto.

Además de las variables de instancia hay **variables de clase**, las cuales se aplican a la clase y a todas sus instancias. Por ejemplo, el número de ruedas de un automóvil es el mismo cuatro, para todos los automóviles.

Tema 2. Clases y Objetos

Clases y objetos

☐ Comportamiento

El comportamiento de los objetos de una clase se implementa mediante funciones miembro o **métodos**. Un método es un conjunto de instrucciones que realizan una determinada **tarea** y son similares a las funciones de los lenguajes estructurados.

Del mismo modo que hay variables de instancia y de clase, también hay **métodos de instancia y de clase**. En el primer caso, un objeto llama a un método para realizar una determinada tarea, en el segundo, el método se llama desde la propia clase.

Tema 2. Clases y Objetos

Clases y objetos

Cuando se escriben los programas orientados a objetos, primero es necesario definir las clases. Cuando el programa está en ejecución, se crean los objetos de esas clases, que van a llevar a cabo tareas. Para indicar a una clase u objeto que realice una tarea es necesario enviarle un *mensaje*.

Por ejemplo, a la clase cuenta, le puedo *mandar un mensaje* "nuevo" para crear una instancia u objeto de esa clase. Igualmente a un objeto de la clase cuenta le puedo *enviar el mensaje* "depositar" para hacer un depósito de 100 euros. Es decir, enviamos mensajes para que realicen tareas.

Tema 2. Clases y Objetos

Clases y objetos

Para que la clase u objeto entienda o procese el mensaje, debe de estar programado, es decir, debe de haber una secuencia de instrucciones que se relacionen con ese mensaje, precisamente esa secuencia de instrucciones es lo que se conoce como **método** y existen métodos definidos para una clase (**métodos de clase**) y métodos definidos para objetos (**método de instancia.**) Por supuesto, en ambos casos, pueden necesitar el paso de parámetros o argumentos.

Diferencias

Método de instancia: Lo definiremos cuando tengamos una tarea que afecte a una instancia u objeto individual.

Método de clase: Se definirá cuando tengamos una tarea que afecte a todas las instancias.

Tema 2. Clases y Objetos

Clases y objetos

De manera análoga a los métodos, que hemos definido de clase y de instancia, podemos definir valores de datos de clase y valores de datos de instancia.

Diferencias

Valores de datos de instancia: Contienen información propia de cada objeto.

Valores de datos de clase: Contiene información compartida por todas las instancias.

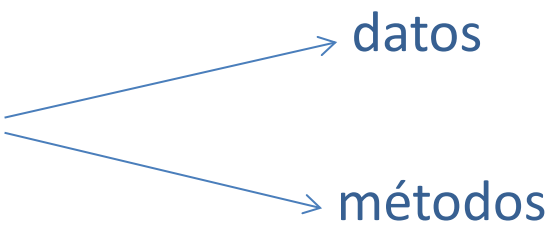
Tema 2. Clases y Objetos

Clases y objetos

☐ Declaración de una clase

Para crear una clase se utiliza la palabra reservada **class** y a continuación el nombre de la clase. La definición de la clase se pone entre las llaves de apertura y cierre. El nombre de la clase empieza por letra mayúscula.

```
<modificador> class <nombre_clase> {  
    <declaración de miembros de clase>  
}
```



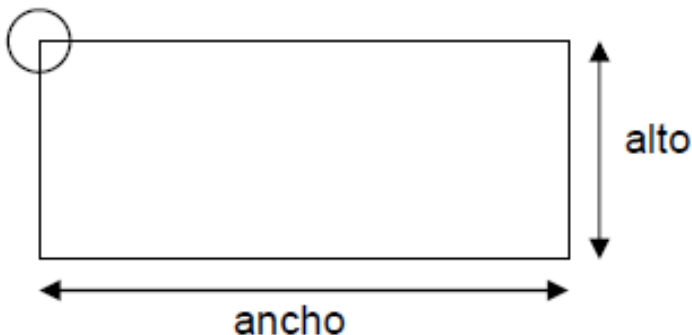
Tema 2. Clases y Objetos

➡ Clases y objetos

Ejemplo 1: Crearemos una clase denominada *Rectangulo*, que describa las características comunes a esas figuras planas.

Atributos

1. Origen del rectángulo: el origen es la posición de la esquina superior izquierda del rectángulo.
2. Dimensiones del rectángulo: ancho y alto.



```
public class Rectangulo {  
    private int x;  
    private int y;  
    private int ancho;  
    private int alto;  
}
```

Atributos

Tema 2. Clases y Objetos

Clases y objetos

Comportamiento

Vamos a añadir operaciones a la clase Rectangulo. Nos interesa:

1. Calcular el área.
2. Desplazar el origen.
3. Saber si contiene en su interior un punto determinado del plano.

Método para el cálculo del área:

No es necesario a ese método pasarle ni el ancho ni el alto del rectángulo, porque son directamente accesibles (esto es, ya están declarados).

```
public int calcularArea() {  
    return ancho * alto;  
}
```

Tema 2. Clases y Objetos

Clases y objetos

Método que desplaza horizontalmente dx y verticalmente dy:

Queremos que desplace al rectángulo horizontalmente una distancia de dx y verticalmente dy. Para ello hacemos lo siguiente:

```
public void desplazar(int dx, int dy) {  
    x += dx;  
    y += dy;  
}
```

Método que determina si un punto está o no en el interior de un rectángulo:

Necesitamos conocer las coordenadas del punto: x_1, y_1 .

Para que el punto (x_1, y_1) esté dentro del rectángulo debe cumplir:

$$x_1 > x \quad \wedge \quad x_1 < x + \text{ancho}$$
$$y_1 > y \quad \wedge \quad y_1 < y + \text{alto}$$


Tema 2. Clases y Objetos

Clases y objetos

```
public boolean estaDentro(int x1, int y1) {  
    if (x1 > x && x1 < x + ancho && y1 > y && y1 < y + alto)  
        return true;  
    else  
        return false;  
}
```

Tema 2. Clases y Objetos

Métodos básicos

-  ☐ Constructores
- ☐ Métodos de acceso y modificadores
- ☐ toString
- ☐ equals
- ☐ compareTo

Tema 2. Clases y Objetos

Métodos básicos

Algunos métodos son comunes a todas las clases. En esta sección los estudiaremos

☐ Constructores

Son métodos que controlan cómo se **crea e inicializa** un objeto. Tienen el mismo nombre que la clase. Gracias a la sobrecarga, se pueden definir diversos constructores.

Cuando se hace una llamada al constructor, se crea un objeto que se sitúa en memoria e inicializa los atributos o miembros de datos declarados en la clase.

Tema 2. Clases y Objetos

☐ Constructores

Se suele hablar de 2 tipos de constructores:

- **Constructores por defecto:** son aquellos que no tienen argumentos, es decir, no se les pasa ningún parámetro.
- **Constructores generales o explícitos:** son aquellos a los que les pasamos parámetros.

Tema 2. Clases y Objetos

➤ □ Constructores

Constructor por defecto

```
public Rectangulo() {  
    super();  
    x = 0;  
    y = 0;  
    ancho = 0;  
    alto = 0;  
}
```

Constructor general

```
public Rectangulo(int x, int y, int ancho, int alto) {  
    super();  
    this.x = x;  
    this.y = y;  
    this.ancho = ancho;  
    this.alto = alto;  
}
```

Cuando los nombres de los parámetros coinciden con los nombres o identificadores de los atributos se usa **this** para distinguir los atributos del parámetro.

El constructor es tan importante que, si el programador no prepara ninguno, el compilador crea uno por defecto, inicializando las variables de los tipos primitivos a su valor por defecto, y los string y las demás referencias a objetos a null.

Tema 2. Clases y Objetos

➤ □ Constructores

Un constructor de una clase puede llamar a otro constructor previamente declarado, siempre y cuando esté declarado en la misma clase, y se hace usando de nuevo la palabra reservada **this**, en este contexto la palabra **this** sólo puede aparecer en la primera sentencia.

```
public Rectangulo(int x, int y) {  
    this(x, y, 0, 0);  
}
```

Llamada al constructor

```
public Rectangulo(int x, int y, int ancho, int alto) {  
    super();  
    this.x = x;  
    this.y = y;  
    this.ancho = ancho;  
    this.alto = alto;  
}
```

Tema 2. Clases y Objetos

Para usar un objeto en un programa se debe de declarar, crear y enviar mensajes

Declaración de un objeto

```
<nombre de clase> <nombre de objeto>;
```

Ejemplo: Rectangulo rect;

Creación de un objeto

```
<nombre de objeto> = new <nombre clase> (<argumentos>)
```

Ejemplo: rect = **new** Rectangulo(1, 1, 1, 1);

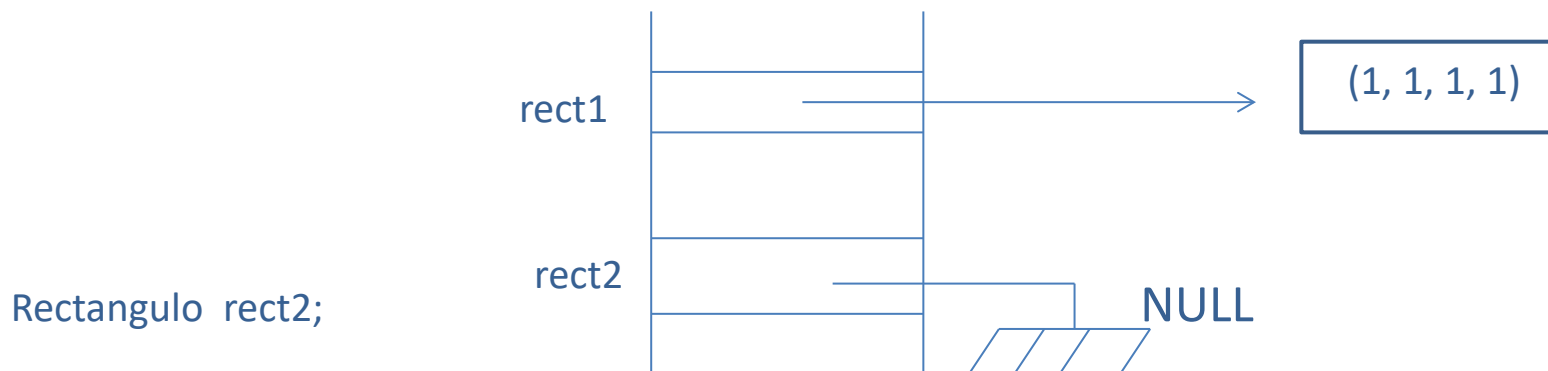
Tema 2. Clases y Objetos

Declaración + creación de un objeto

<nombre de clase> <nombre de objeto> = **new** <nombre clase> (<argumentos>)

Ejemplo: Rectangulo rect1= **new** Rectangulo(1, 1, 1, 1);

rect1 tiene la dirección de memoria donde se almacena el objeto y se dice que rect1 apunta al objeto o referencia al objeto.



Tema 2. Clases y Objetos

Envío de mensajes a un objeto

Una vez creado un objeto, para llevar a cabo una determinada tarea, por ejemplo obtener su área, es necesario enviarle un mensaje. La sintaxis de **envío de mensajes** es:

`<nombre de objeto> . <nombre del método> (< parámetros>)`

Por ejemplo:

```
int medidaArea1 = rect1.calcularArea();  
rect1.desplazar(10, 20);  
boolean esta = rect1.estaDentro(10,10);
```

Tema 2. Clases y Objetos

➤ □ Métodos de acceso y modificadores

Los atributos de una clase se declaran normalmente como privados. En consecuencia, los métodos que no pertenecen a la clase no pueden acceder directamente a ellos. En ocasiones nos gustaría examinar el valor de un atributo e incluso cambiarlo. Una posibilidad es declarar los atributos como públicos. Sin embargo esta elección viola el principio de ocultamiento de información. Para evitarlo proporcionaremos métodos para examinar y cambiar los atributos.

Un método que examina, pero no cambia el estado de un objeto es un **método de acceso**.

Un método que cambia el estado de un objeto es un **método modificador**

Tema 2. Clases y Objetos

➤ □ Métodos de acceso y modificadores

Se conocen como **Getters** (de acceso) y **Setters** (modificadores)

```
public int getAncho() {  
    return ancho;  
}  
  
public void setAncho(int ancho) {  
    this.ancho = ancho;  
}
```

Ejemplo de uso: `int anchura = rect1.getAncho();`
`rect1.setAncho(10);`

Tema 2. Clases y Objetos

➤ □ toString

Es un método que devuelve un String mostrando el estado del objeto, es decir sus atributos.

```
public String toString() {  
    return "Rectangulo x=" + x + ", y=" + y + ", ancho=" + ancho  
        + ", alto=" + alto;  
}
```

Ejemplo de uso: `System.out.println(rect1.toString());`

Tema 2. Clases y Objetos

➤ `equals`

Es un método que se utiliza para comprobar si dos referencias describen el mismo valor (compara si dos objetos son iguales o no).

```
public boolean equals(Object obj) {  
    Rectangulo otro = (Rectangulo) obj;  
    return x == otro.x && y == otro.y && ancho == otro.ancho  
        && alto == otro.alto;  
}
```

Ejemplo de uso:

```
if(rect1.equals(rect2)) → parámetro explícito  
    System.out.println("Mismos rectangulos");  
else  
    System.out.println("Distintos rectangulos");
```

← parámetro implícito

Tema 2. Clases y Objetos

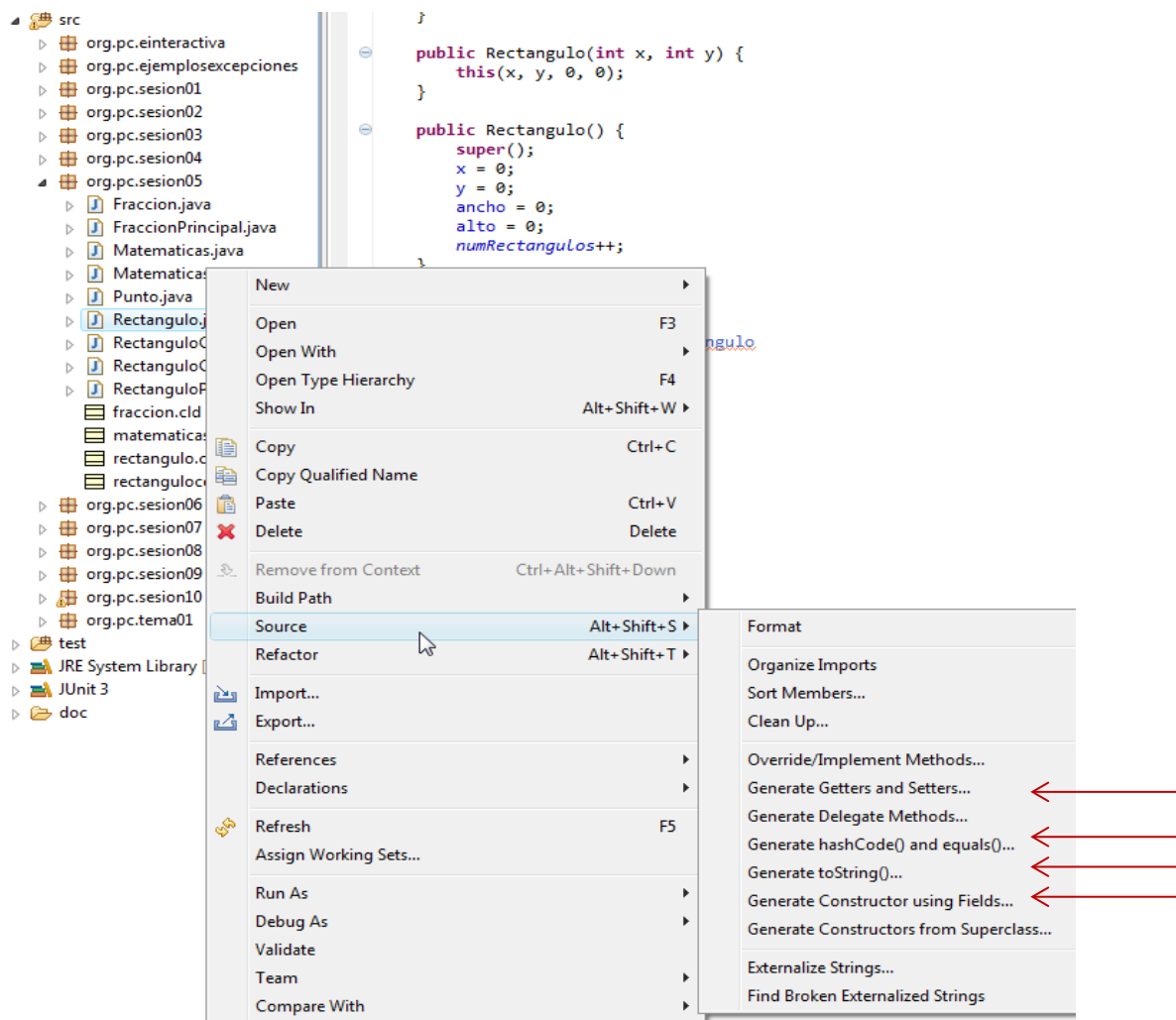
➤ ☐ compareTo

Es un método que se utiliza para comparar, por algún criterio, dos objetos. Devuelve **0** si los objetos son iguales, **1** si el primer objeto es mayor y **-1** si el primer objeto es menor.

```
public int compareTo(Object obj) {  
    Rectangulo otro = (Rectangulo) obj;  
    if (calcularArea() == otro.calcularArea())  
        return 0;  
    else if (calcularArea() < otro.calcularArea())  
        return -1;  
    else  
        return 1;  
}
```

Ejemplo de uso: **if (rect1.compareTo(rect2) == 0)**
 System.out.println("Rectangulos iguales");
else if (rect1.compareTo(rect2) == -1)
 System.out.println("El primer rectángulo es menor que el segundo");
else
 System.out.println("El primer rectángulo es mayor que el segundo");

Eclipse da la facilidad de escribir de manera automática algunos de estos métodos.



```

1 package org.ip.tema02;
2
3 public class Rectangulo {
4     private int x;
5     private int y;
6     private int ancho;
7     private int alto;
8     private static int numRectangulos = 0;
9
10    public Rectangulo(int x, int y, int ancho, int alto) {
11        super();
12        this.x = x;
13        this.y = y;
14        this.ancho = ancho;
15        this.alto = alto;
16        numRectangulos++;
17    }
18
19    public Rectangulo(int x, int y) {
20        this(x, y, 0, 0);
21    }
22
23    public int getX() {
24        return x;
25    }
26    public void setX(int x) {
27        this.x = x;
28    }
29    public int getY() {
30        return y;
31    }
32    public void setY(int y) {
33        this.y = y;
34    }

```

```

35    public int getAncho() {
36        return ancho;
37    }
38    public void setAncho(int ancho) {
39        this.ancho = ancho;
40    }
41    public int getAlto() {
42        return alto;
43    }
44    public void setAlto(int alto) {
45        this.alto = alto;
46    }
47    public static int getNumRectangulos() {
48        return numRectangulos;
49    }
50
51    @Override
52    public String toString() {
53        return "Rectangulo [x=" + x + ", y=" + y + ", ancho=" + ancho
54            + ", alto=" + alto + "]";
55    }
56
57    @Override
58    public boolean equals(Object obj) {
59        if (this == obj)
60            return true;
61        if (obj == null)
62            return false;
63        if (getClass() != obj.getClass())
64            return false;
65        Rectangulo otro = (Rectangulo) obj;
66        return x == otro.x && y == otro.y && ancho == otro.ancho
67            && alto == otro.alto;
68    }

```



```
70- /**
71  * Metodo que compara dos rectangulos, devolviendo 0 si son iguales,
72  * -1 si el primero es menor y 1 si el primero es mayor
73  * @param obj
74  * @return un entero 0, -1, 1
75  */
76- public int compareTo(Object obj) {
77     Rectangulo otro = (Rectangulo) obj;
78
79     if (calcularArea() == otro.calcularArea())
80         return 0;
81     else if (calcularArea() < otro.calcularArea())
82         return -1;
83     else
84         return 1;
85 }
86
87- public void desplazar(int dx, int dy) {
88     x = x + dx;
89     y = y + dy;
90 }
91
92- public boolean estaDentro(int x1, int y1) {
93     if (x1 > x && x1 < x + ancho && y1 > y && y1 < y + alto)
94         return true;
95     else
96         return false;
97 }
98
99- public int calcularArea() {
100     return ancho * alto;
101 }
102 }
```

```

1 package org.ip.tema02;
2
3 public class RectanguloPrincipal {
4
5     public static void main(String[] args) {
6         Rectangulo rect1 = new Rectangulo(10, 15, 20, 5);
7         Rectangulo rect2 = new Rectangulo(7, 15, 7, 5);
8         Rectangulo rect3 = new Rectangulo(10, 15, 20, 20);
9
10        int altura = rect1.getAlto();
11        if (altura < 10)
12            System.out.println("Rectángulo 1 pequeño");
13        System.out.println("Rectángulo 1 " + rect1.toString());
14        if (rect1.equals(rect2))
15            System.out.println("Mismos rectángulos 1 y 2");
16        else
17            System.out.println("Distintos rectángulos 1 y 2");
18        if (rect1.equals(rect3))
19            System.out.println("Mismos rectángulos 1 y 3");
20        else
21            System.out.println("Distintos rectángulos 1 y 3");
22        System.out.println("El número de rectángulos creados es "
23            + Rectangulo.getNumRectangulos());
24        rect1.desplazar(7, 9);
25        System.out.println("Rectángulo 1 " + rect1.toString());
26        if (rect1.compareTo(rect3) == 0)
27            System.out.println("Rectángulos 1 y 3 con igual área");
28        else if (rect1.compareTo(rect3) == -1)
29            System.out.println("Rectángulo 1 tiene menor área que 3");
30        else
31            System.out.println("Rectángulo 1 tiene mayor área que 3");
32        if (rect1.estaDentro(20, 30))
33            System.out.println("El punto (20, 30) está dentro del rectángulo 1");
34        else
35            System.out.println("El punto (20, 30) no está dentro del rectángulo 1");
36    }
37 }
38

```

Salida

```

Rectángulo 1 pequeño
Rectángulo 1 Rectangulo [x=10, y=15, ancho=20, alto=5]
Distintos rectángulos 1 y 2
Distintos rectángulos 1 y 3
El número de rectángulos creados es 3
Rectángulo 1 Rectangulo [x=17, y=24, ancho=20, alto=5]
Rectángulo 1 tiene menor área que 3
El punto (20, 30) no está dentro del rectángulo 1

```

Tema 2. Clases y Objetos

Paquetes

Se usan para organizar una serie de clases relacionadas. Cada paquete consta de un conjunto de clases.

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 2. 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 2. 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 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 2. Clases y Objetos

Algunos métodos de la clase **Scanner**

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

Previamente el objeto entrada deberá estar instanciado o creado.

```
1 package org.ip.tema02;
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 2. 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**

(comportamiento de los objetos de esa clase)

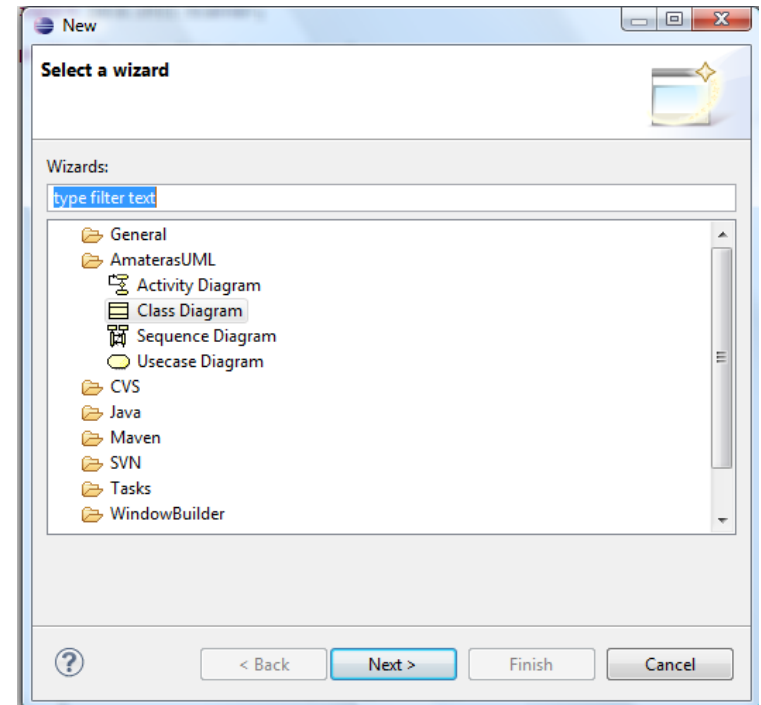
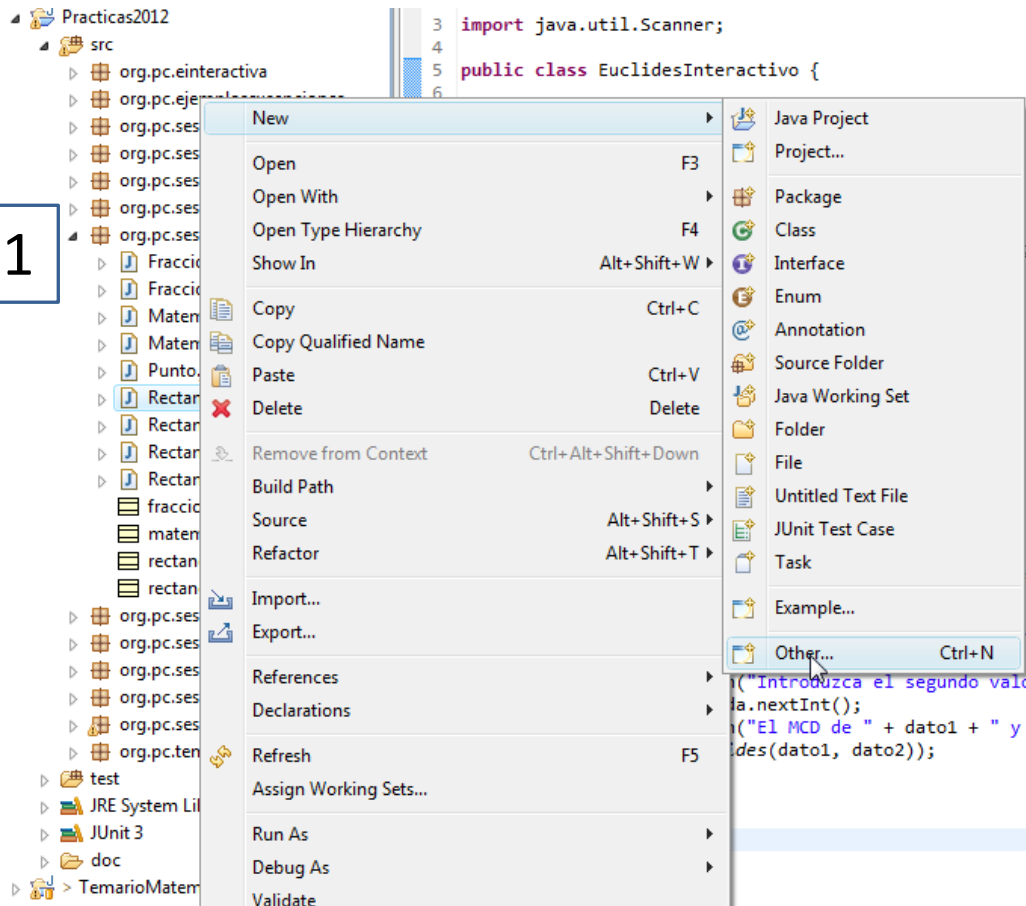
En **Eclipse** podemos generar estos diagramas , llamados de clase a partir de un plug-in **Amateras**.

Tema 2. Clases y Objetos

Generación de diagramas de clase con Amateras

2

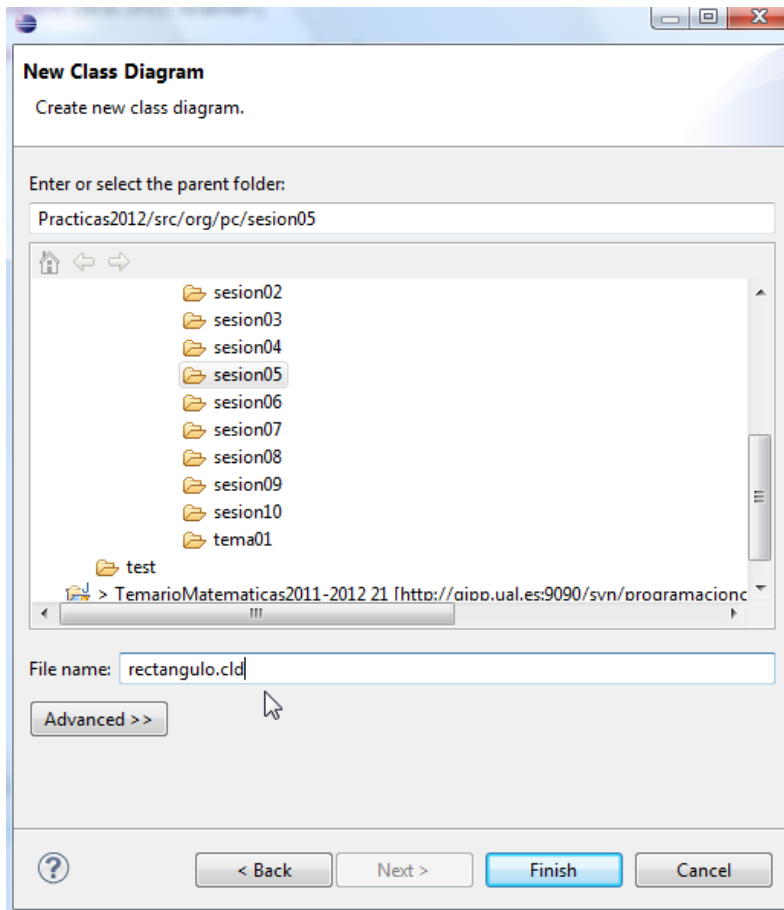
1



Tema 2. Clases y Objetos

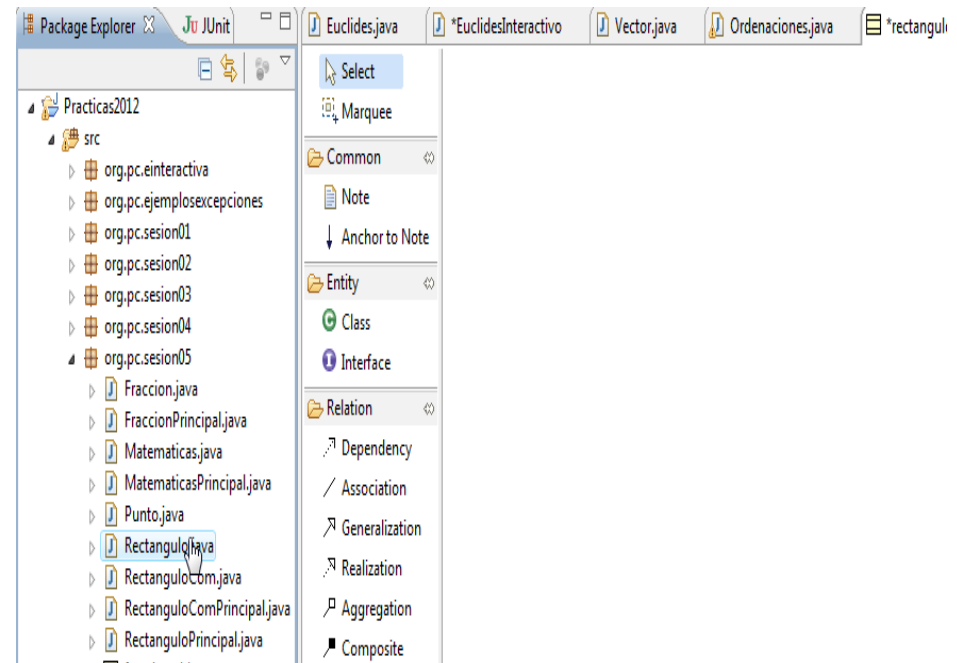
Generación de diagramas de clase con Amateras

3



Arrastramos la clase Rectangulo a la ventana

4



Tema 2. Clases y Objetos

Generación de diagramas de clases con Amateras

5

org.ip.tema02.Rectangulo
<ul style="list-style-type: none">□ x: int□ y: int□ ancho: int□ alto: int□ <u>numRectangulos: int</u>
<ul style="list-style-type: none">● Rectangulo(x: int, y: int, ancho: int, alto: int)● Rectangulo(x: int, y: int)● getX(): int● setX(x: int): void● getY(): int● setY(y: int): void● getAncho(): int● setAncho(ancho: int): void● getAlto(): int● setAlto(alto: int): void● <u>getNumRectangulos(): int</u>● toString(): String● equals(obj: Object): boolean● compareTo(obj: Object): int● desplazar(dx: int, dy: int): void● estaDentro(x1: int, y1: int): boolean● calcularArea(): int

Tema 2. 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.tema02;
2
3 public class TestCirculo1 {
4
5     public static void printCirculo(Circulo1 c) {
6         System.out.printf("El área del círculo 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 }

```

org.ip.tema02.Circulo1
radio: double
numCirculos: int
Circulo1(radio: double)
getRadio(): double
setRadio(radio: double): void
getNumCirculos(): int
getArea(): double

Salida

El área del círculo 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.tema02;
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 }

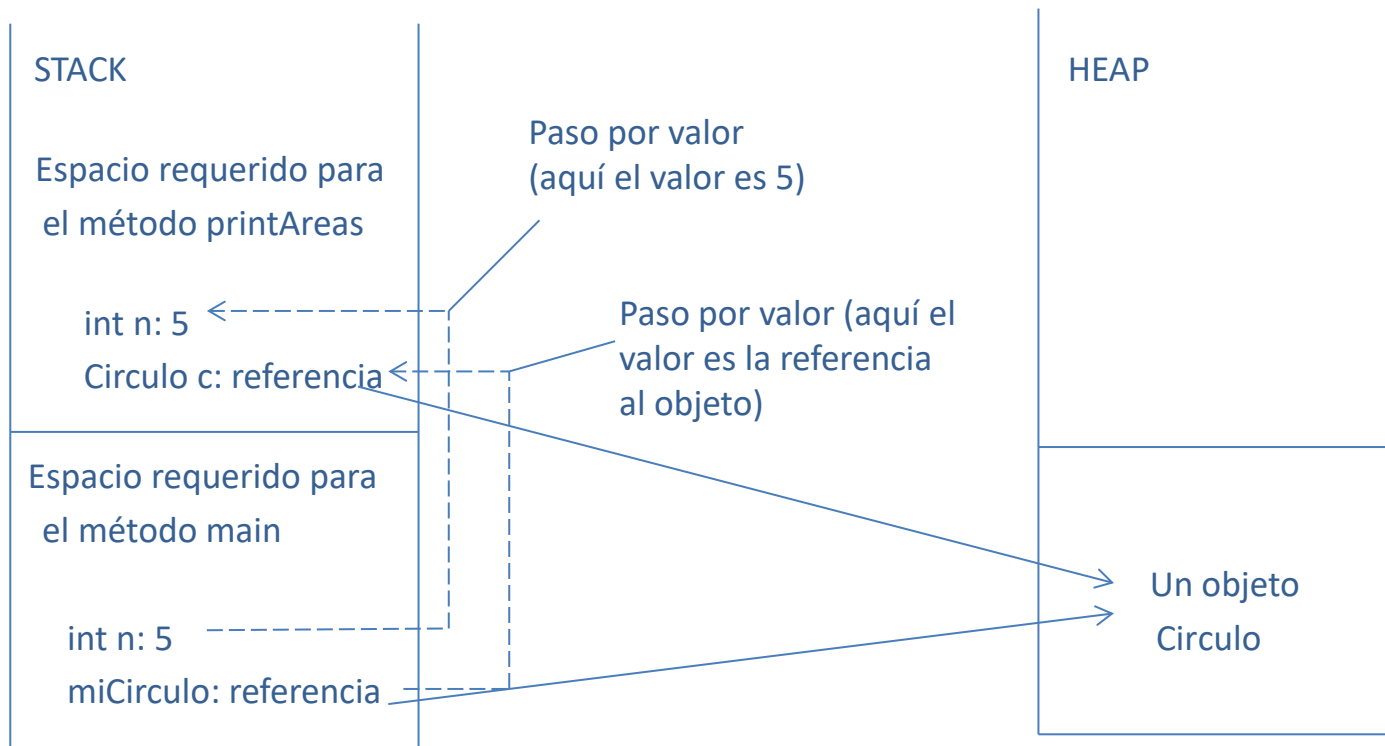
```

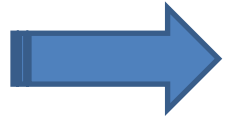
Salida

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 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

Cuando se pasa como parámetro un tipo de **dato primitivo**, el *parámetro real* pasa **el valor** al *parámetro formal* y si éste se modifica en el método, el *parámetro real* no se ve afectado.

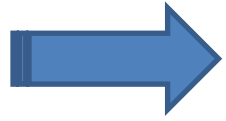
Cuando se pasa como *parámetro un tipo referencia*, se pasa la **referencia al objeto**. En este caso, **c** (*parámetro formal*) contiene la referencia al objeto que también es referenciado por **miCirculo** (*parámetro real*). Por ello, un cambio en las propiedades del objeto **c** en el método **printAreas** 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*, el objeto referenciado en el método es el mismo objeto que el pasado como parámetro real.



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 primitivos de datos 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, claramente, los objetos no se pasan por valor.

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:

```
package org.ip.tema02;
```

```
public class PasaObjetosMetodos {
```

```
    private static void cambiarNombre(Persona p) {
```

```
        //p = new Persona();           // nuevo valor al parametro           }
```

```
        p.setNombre("Antonio");         // aquí tendrá otro valor
```

```
        System.out.println("El nombre en el método es: " + p.getNombre());
```

```
    }
```

```
    public static void main(String[] args) {
```

```
        Persona p = new Persona();    // creamos un objeto
```

```
        p.setNombre("Juan");            // le asignamos un valor
```

```
        System.out.println("El nombre antes del método es: " + p.getNombre());
```

```
        cambiarNombre(p);               // llamamos a un método para que cambie de valor
```

```
        System.out.println("El nombre después del método es: " + p.getNombre());
```

```
    }
```

```
}
```

```
package org.ip.tema02;
```

```
public class Persona {
```

```
    private String nombre;
```

```
    public String getNombre() {
```

```
        return nombre;
```

```
    }
```

```
    public void setNombre(String nombre) {
```

```
        this.nombre = nombre;
```

```
    }
```

Tema 2. 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**:

org.ip.tema02.Punto
<ul style="list-style-type: none"> x: int y: int
<ul style="list-style-type: none"> Punto(x: int, y: int) toString(): String equals(obj: Object): boolean getX(): int setX(x: int): void getY(): int setY(y: int): void desplazar(dx: int, dy: int): void

```

1 package org.ip.tema02;
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 2. Clases y Objetos

Composición

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

```

1 package org.ip.tema02;
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 }

```

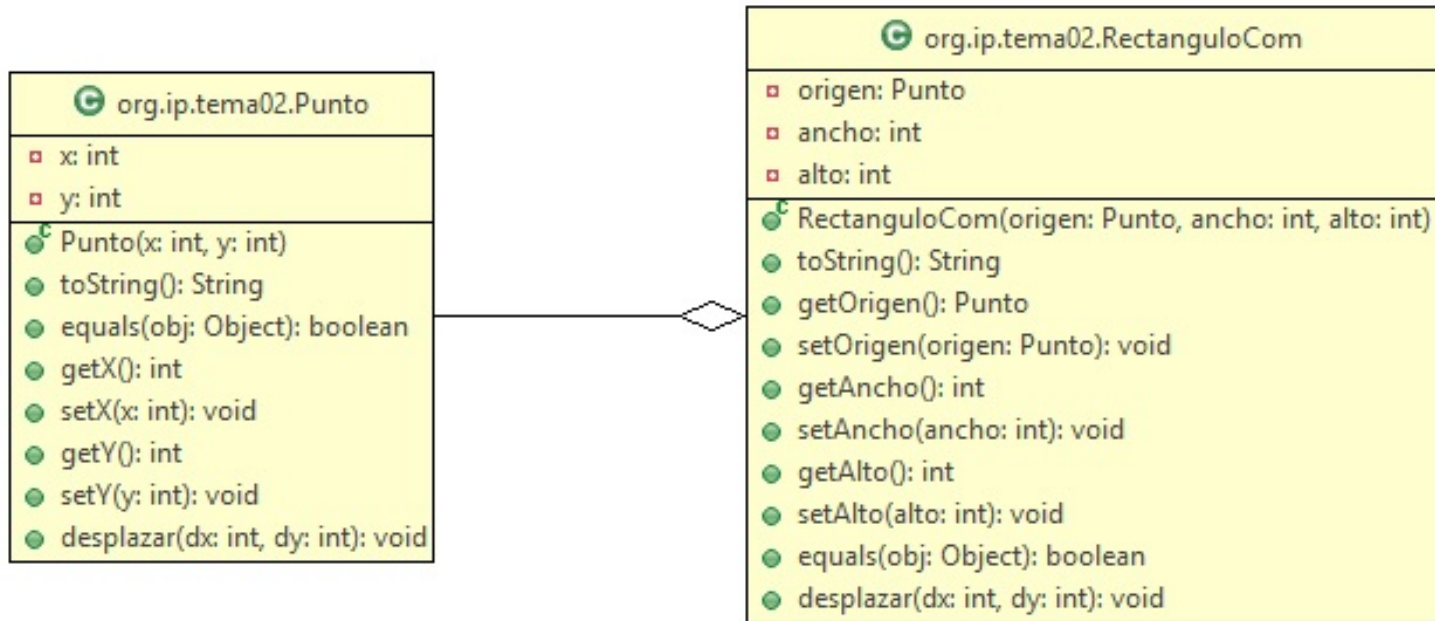
org.ip.tema02.RectanguloCom
<ul style="list-style-type: none"> origen: Punto ancho: int alto: int
<ul style="list-style-type: none"> RectanguloCom(origen: Punto, ancho: int, alto: int) toString(): String getOrigen(): Punto setOrigen(origen: Punto): void getAncho(): int setAncho(ancho: int): void getAlto(): int setAlto(alto: int): void equals(obj: Object): boolean desplazar(dx: int, dy: int): void

Tema 2. Clases y Objetos



Composición

Diagrama de clases



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

Tema 2. Clases y Objetos



Composición

Ejemplo de uso

```
1 package org.ip.tema02;
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 2. Clases y Objetos

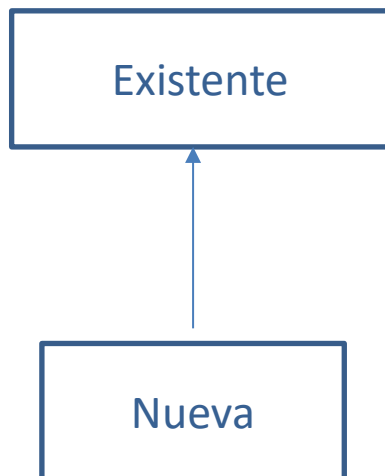
➡ Herencia

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 2. Clases y Objetos



Herencia

Declaración de una clase derivada

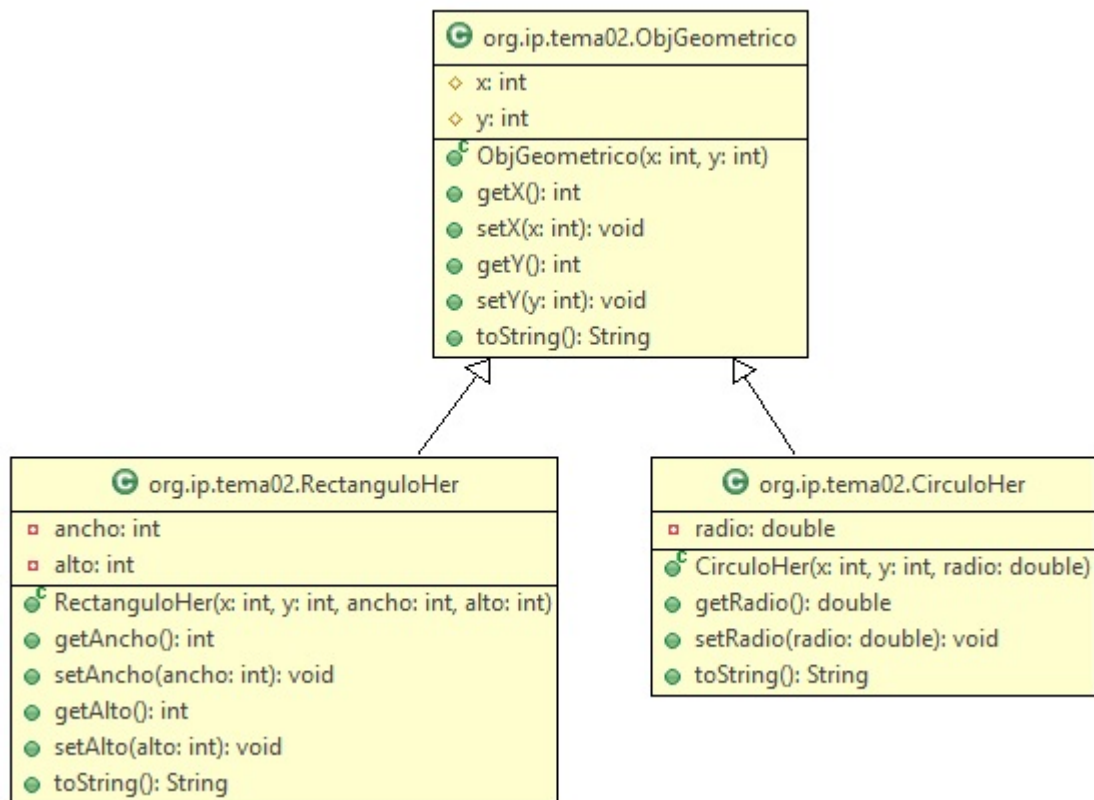
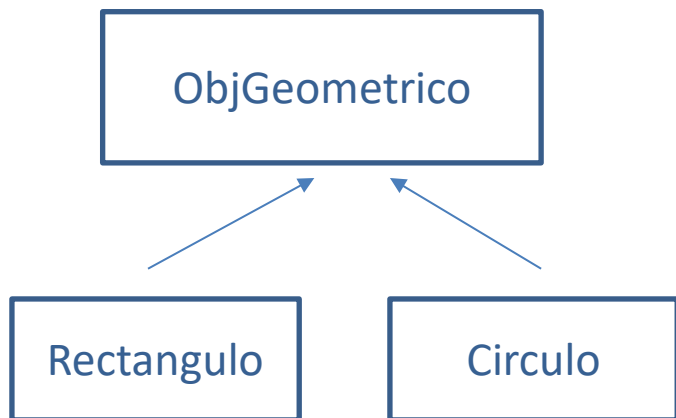
```
<modificadores> class <nombre clase derivada> extends <nombre clase base> {  
    // atributos, métodos  
}
```

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

Tema 2. Clases y Objetos

➡ Herencia

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



Tema 2. 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 2. Clases y Objetos

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

Tema 2. Clases y Objetos

Herencia

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

```

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 }

```

Tema 2. 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

```

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 2. 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

Tema 2. Clases y Objetos

Clases abstractas e Interfaces

Clases abstractas

Cuando se hace uso de herencia los métodos pueden:

- Permanecer invariantes a lo largo de la jerarquía de clases. (modificador **final**)
- Necesitar ser modificados (redefinirlos).
- Una tercera posibilidad es que el método tenga sentido para las clases derivadas y que deba por tanto implementarse en ellas mientras que su implementación no tiene sentido en la clase base. Este sería el caso de los *métodos abstractos*.

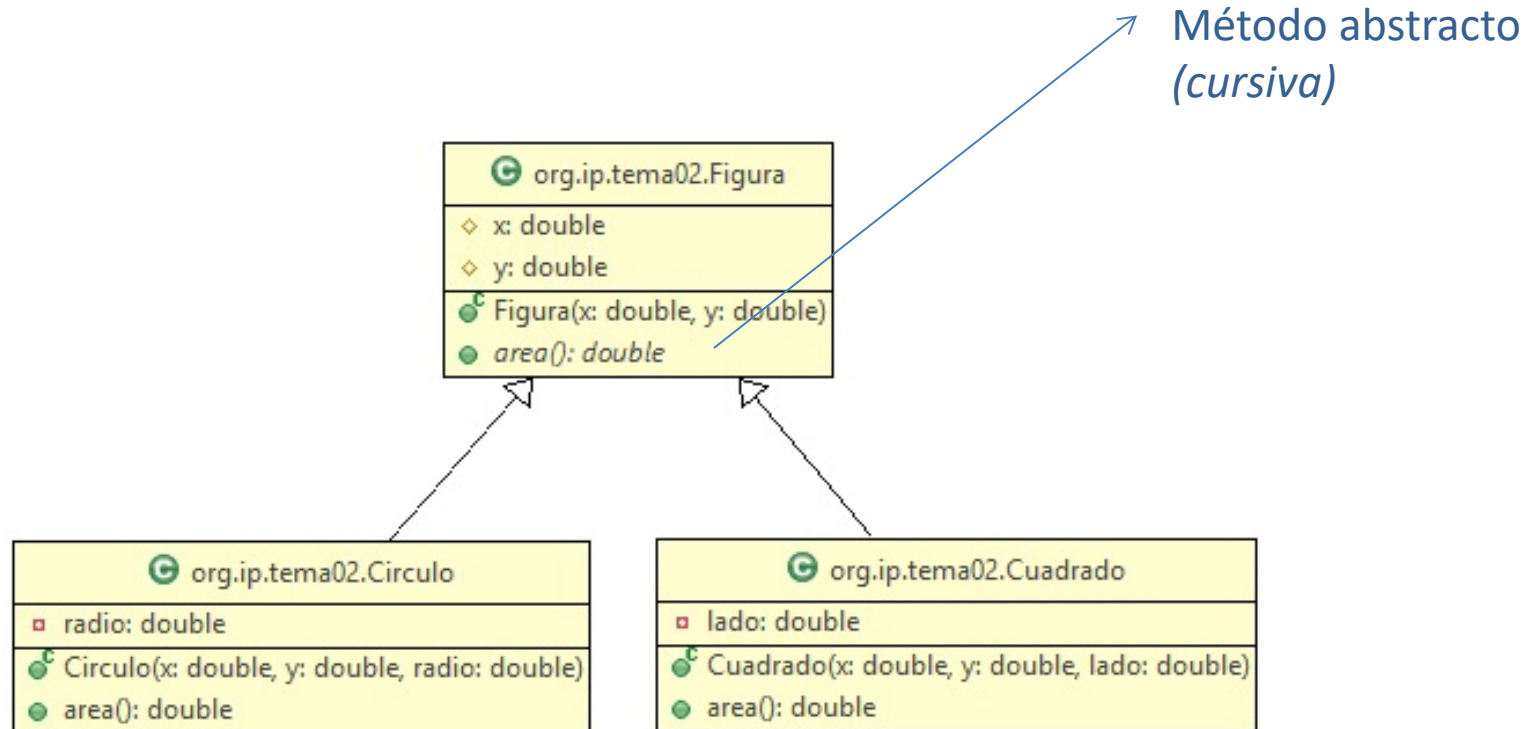
Tendremos una **clase abstracta** en cuanto al menos uno de sus métodos sea abstracto, es decir no tenga implementación. En Java, el método abstracto se declara en la clase base con la palabra reservada **abstract** y se debe implementar en las clases derivadas.

Una **clase abstracta** es una clase que no permite instanciar objetos, se usa únicamente para definir subclases.

Tema 2. Clases y Objetos

Clases abstractas e Interfaces

Ejemplo clase abstracta



Tema 2. Clases y Objetos

Clases abstractas e Interfaces

Clases derivadas

Ejemplo clase abstracta

Clase base

```
package org.ip.tema02;

public abstract class Figura {
    protected double x;
    protected double y;

    public Figura(double x, double y) {
        super();
        this.x = x;
        this.y = y;
    }

    public abstract double area();
}
```

Método abstracto

Implementaciones métodos abstractos

```
package org.ip.tema02;

public class Circulo extends Figura {
    private double radio;

    public Circulo(double x, double y, double radio) {
        super(x, y);
        this.radio = radio;
    }

    @Override
    public double area() {
        return Math.PI * radio * radio;
    }
}

package org.ip.tema02;

public class Cuadrado extends Figura {
    private double lado;

    public Cuadrado(double x, double y, double lado) {
        super(x, y);
        this.lado = lado;
    }

    @Override
    public double area() {
        return lado * lado;
    }
}
```

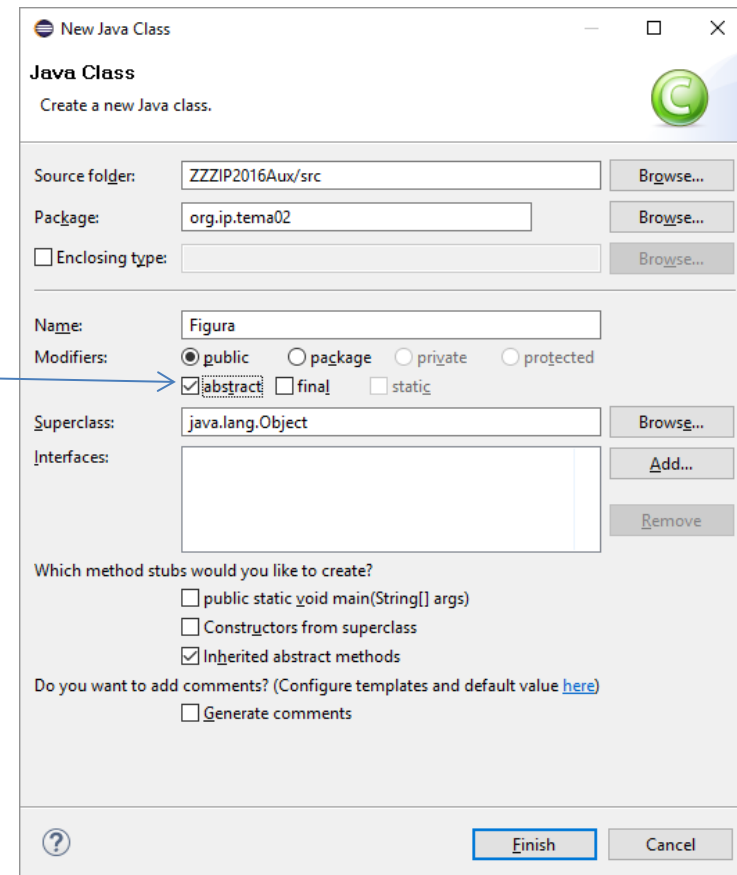
Tema 2. Clases y Objetos

Clases abstractas e Interfaces

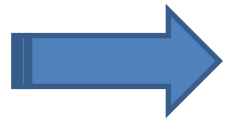
En Eclipse

Para crear una clase abstracta: **New → Class**

Además seleccionaremos el modificador **abstract** para indicar que esa clase es abstracta



Tema 2. Clases y Objetos



Clases abstractas e Interfaces

Interfaces

Es una clase que no contiene ningún detalle de implementación.

La diferencia entre una clase abstracta y una interfaz es que aunque ambas especifican como deben comportarse las clases derivadas, la interfaz no puede dar ningún detalle.

Las interfaces se declaran con la palabra reservada **interface** de manera similar a como se declaran las clases abstractas.

En la declaración solo puede aparecer declaraciones de métodos (cabeceras, sin implementación) y definiciones de constantes.

Su utilidad reside en definir unos requisitos mínimos que debe cumplir cualquier objeto que creamos a partir de una clase que la implemente.

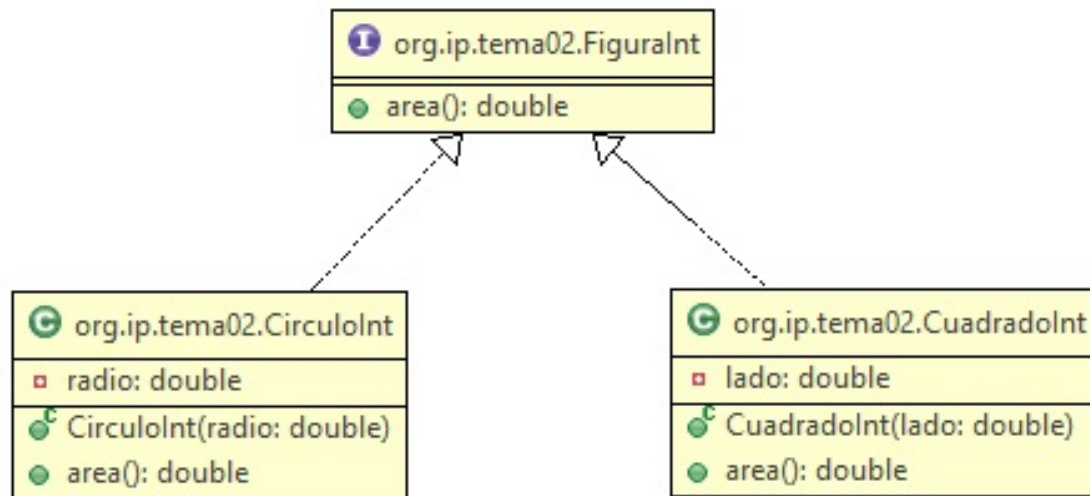
Una interfaz **declara** pero **no implementa** las acciones mínimas que poseerá un objeto.

Para indicar que una clase implementa una interfaz se utiliza la palabra reservada **implements**.

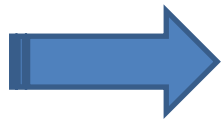
Tema 2. Clases y Objetos

➔ Clases abstractas e Interfaces

Ejemplo interfaz



Tema 2. Clases y Objetos



Clases abstractas e Interfaces

Clases que implementan la interfaz

Ejemplo interfaz

Interfaz

```
package org.ip.tema02;

public interface FiguraInt {
    public double area();
}
```

Palabras reservadas:

interface (Interfaz)

implements (Clases que implementan la Interfaz)

```
package org.ip.tema02;

public class CirculoInt implements FiguraInt {
    private double radio;

    public CirculoInt(double radio) {
        super();
        this.radio = radio;
    }

    @Override
    public double area() {
        return Math.PI * radio * radio;
    }
}

package org.ip.tema02;

public class CuadradoInt implements FiguraInt {
    private double lado;

    public CuadradoInt(double lado) {
        super();
        this.lado = lado;
    }

    @Override
    public double area() {
        return lado * lado;
    }
}
```

Tema 2. Clases y Objetos



Clases abstractas e Interfaces

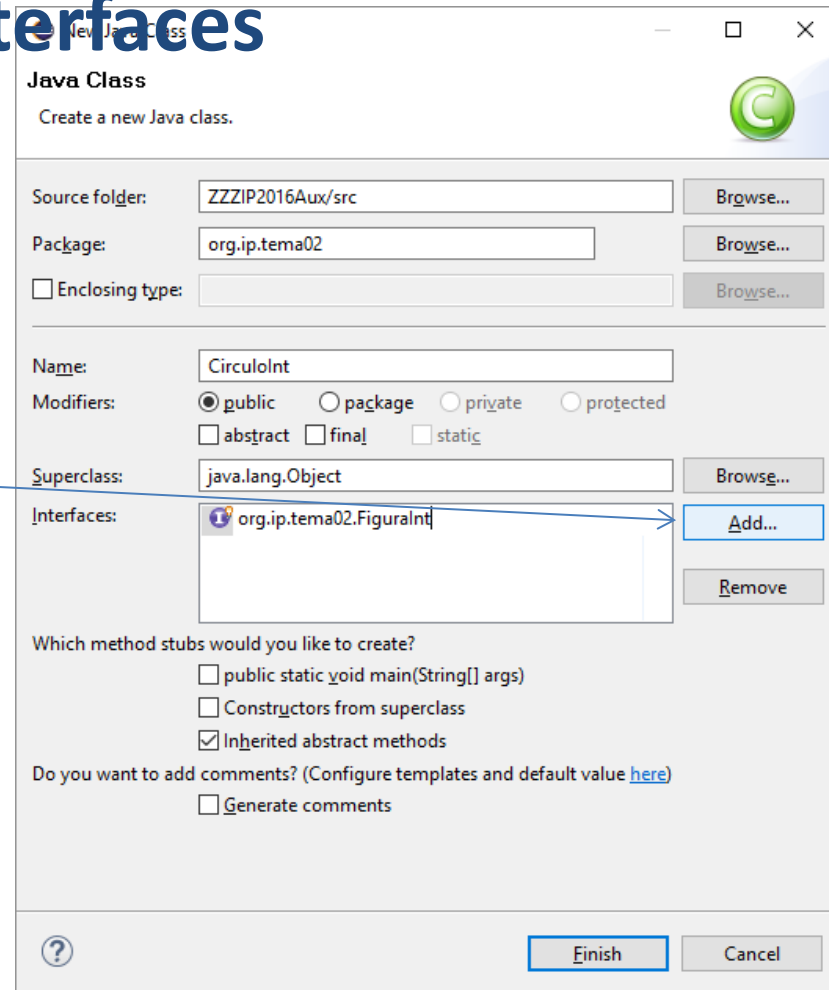
En Eclipse

Para crear una interfaz: **New → Interface**

Para crear una clase que implemente

una interfaz: **New → Class** y después

Add y seleccionamos la interfaz que implementa



Tema 2. Clases y Objetos



Clases abstractas e Interfaces

Una clase puede implementar varias interfaces simultáneamente pero solo puede heredar de una clase (herencia simple, múltiple interfaces)

Ejemplo:

```
public class Cuadrado extends Figura implements Dibujable, Rotable
```

Esto quiere decir que la clase Cuadrado, heredaría de la clase Figura e implementa las interfaces Dibujable y Rotable.

Tema 2. Clases y Objetos



Clases abstractas e Interfaces

La interfaz **Comparable**. Ejemplo importante.

Es una interface para comparar objetos, definida en el paquete **java.lang**. Tiene un único método **compareTo** cuya implementación, en cualquier clase que implemente la interface, determinará el orden para comparar dos objetos (= 0, < 0, > 0 cuando los objetos sean iguales o el primero menor que el segundo o el primero mayor que el segundo respectivamente).

```
package java.lang
```

```
public interface Comparable{
```

```
    public int compareTo(Object o);
```

```
}
```


Tema 2. Clases y Objetos



Excepciones

Las excepciones son la manera que ofrece Java de **manejar los errores en tiempo de ejecución**.

Los lenguajes imperativos simplemente detienen la ejecución del programa cuando surge un error.

Las excepciones nos permiten escribir código para manejar el error y continuar (si lo estimamos conveniente) con la ejecución del programa.

Veamos un ejemplo de un programa que obtiene el cociente de dos números enteros.

Tema 2. Clases y Objetos



Excepciones

```
package org.ip.tema02;

import java.util.Scanner;

public class Cociente {

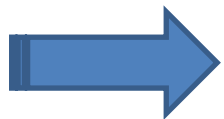
    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);
        System.out.println("Introduzca dos enteros: ");
        int num1 = entrada.nextInt();
        int num2 = entrada.nextInt();
        System.out.println(num1 + " / " + num2 + " es " + (num1 / num2));
    }
}
```

Salidas

```
Introduzca dos enteros:
5 2
5 / 2 es 2
```

```
Introduzca dos enteros:
5 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
at org.ip.tema02.Cociente.main(Cociente.java:12)
```

Tema 2. Clases y Objetos



Excepciones

Una posible solución sería incluir una sentencia **if**

```
package org.ip.tema02;

import java.util.Scanner;

public class CocienteConIf {

    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);

        System.out.println("Introduzca dos enteros: ");
        int num1 = entrada.nextInt();
        int num2 = entrada.nextInt();

        if (num2 != 0)
            System.out.println(num1 + " / " + num2 + " es " + (num1 / num2));
        else
            System.out.println("No se puede hacer una división por cero");
    }
}
```

Salida

Introduzca dos enteros:

5 0

No se puede hacer una división por cero

Esta forma de gestionar los errores es propia de los lenguajes imperativos que no disponen de otros mecanismos.

Tema 2. Clases y Objetos

Excepciones

En lenguajes de programación orientados a objetos como Java, el propio lenguaje incorpora la gestión de errores proporcionando construcciones especiales para ello.

En el ejemplo cociente se produce un error de ejecución (lanza una excepción) al intentar dividir por cero.

Cuando se detecta el error, por defecto se interrumpe la ejecución. Pero podemos evitarlo.

La estructura **try-catch-finally** nos permitirá capturar excepciones, es decir, reaccionar a un error de ejecución.

De este modo podremos imprimir mensajes de error *a la medida* y continuar con la ejecución del programa si consideramos que el error no es demasiado grave.

Para ver cómo funcionan vamos a modificar el ejemplo anterior, pero asegurándonos ahora de que capturamos las excepciones.

Tema 2. Clases y Objetos



Excepciones

```
try {  
    // Código que puede hacer que se eleve la excepción  
}  
  
catch (TipoExcepcion e) {  
    // Gestor de la excepción  
}
```

El comportamiento de Java es el siguiente:

Si en la ejecución del código dentro del bloque **try** se lanza una excepción de tipo `TipoExcepcion` (o descendiente de éste), Java omite la ejecución del resto del código en el bloque **try** y ejecuta el código situado en el bloque **catch** (gestor).

Tema 2. Clases y Objetos



Excepciones

try

El bloque de código donde se prevé que se lance una excepción. Al encerrar el código en un bloque **try** es como si dijéramos: *Prueba a usar estas instrucciones y mira a ver si se produce alguna excepción*. El bloque **try** tiene que ir seguido, al menos, por una cláusula **catch** o una cláusula **finally**.

catch

El código que se ejecuta cuando se lanza la excepción. Controla cualquier excepción que cuadre con su argumento. Se pueden colocar varios **catch** sucesivos, cada uno controlando un tipo de excepción diferente.

finally

Bloque que se ejecuta siempre, haya o no excepción.

Tema 2. Clases y Objetos



Excepciones

El ejemplo quedaría:

```
package org.ip.tema02;

import java.util.Scanner;

public class CocienteConExcepcion {

    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);

        System.out.println("Introduzca dos enteros: ");
        int num1 = entrada.nextInt();
        int num2 = entrada.nextInt();
        try {
            System.out.println(num1 + " / " + num2 + " es " + (num1 / num2));
        } catch (Exception ex) {
            System.out.println("Excepción: un entero "
                + "no se puede dividir por cero");
        }
        System.out.println("La ejecución continua ...");
    }
}
```

Salida

```
Introduzca dos enteros:
5 0
Excepción: un entero no se puede dividir por cero
La ejecución continua ...
```

Tema 2. Clases y Objetos



Excepciones

Podríamos utilizar un método

```
package org.ip.tema02;

import java.util.Scanner;

public class CocienteConMetodo {
    public static int cociente(int numero1, int numero2) {
        return numero1 / numero2;
    }

    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);

        System.out.println("Introduzca dos enteros: ");
        int num1 = entrada.nextInt();
        int num2 = entrada.nextInt();
        try {
            int resultado = cociente(num1, num2);
            System.out.println(num1 + " / " + num2 + " es "
                + resultado);
        } catch (Exception ex) {
            System.out.println("Excepción: un entero "
                + "no se puede dividir por cero");
        }
        System.out.println("La ejecución continua ...");
    }
}
```

Salida

Introduzca dos enteros:

5 0

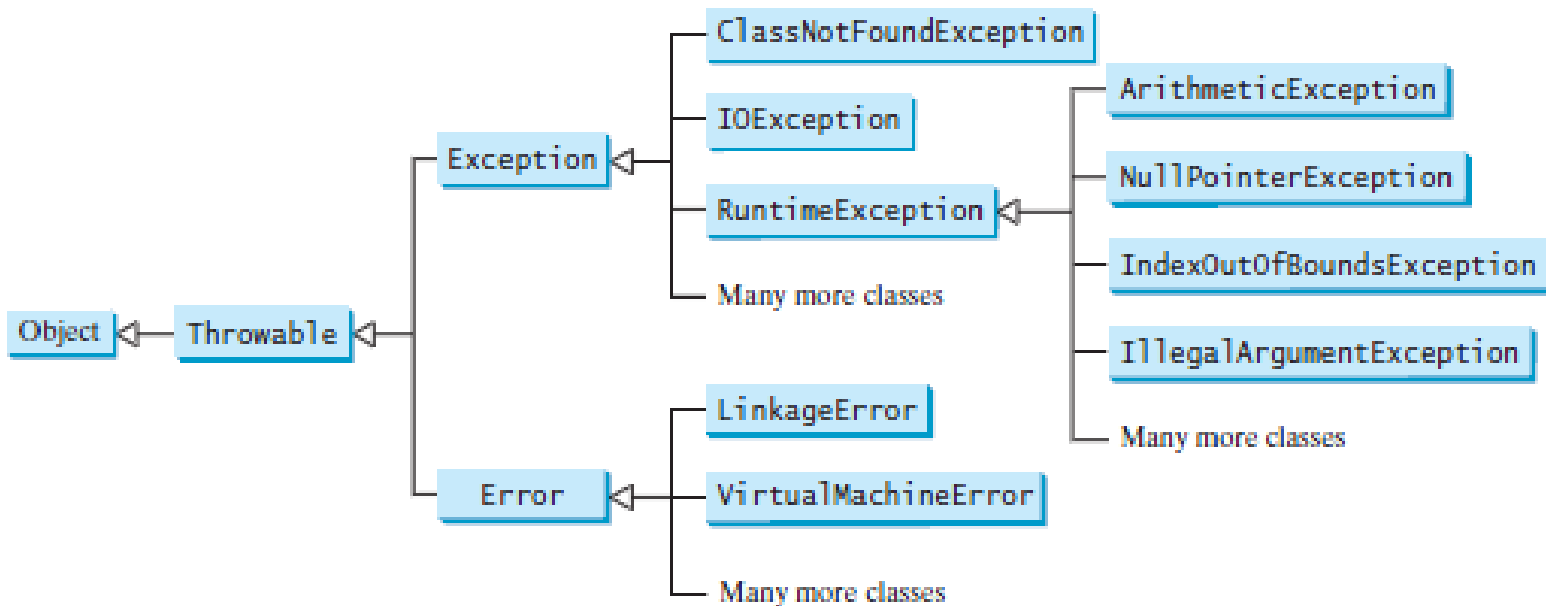
Excepción: un entero no se puede dividir por cero

La ejecución continua ...

Tema 2. Clases y Objetos

➔ Excepciones

Jerarquía de clases para el manejo de excepciones. Excepciones predefinidas



Tema 2. Clases y Objetos



Excepciones

Throwable

Superclase que engloba a todas las excepciones

Error

Representa los errores graves causados por el sistema (JVM, etc.); no son tratados por los programas.

Exception

Define las excepciones que los programas deberían tratar (IOException, ArithmeticException, etc.).

Tema 2. Clases y Objetos

Excepciones

La clase **Exception**

Cuando se lanza una excepción, lo que se hace es activar un ejemplar de **Exception** o de alguna de sus subclases.

Normalmente las clases derivadas nos permiten distinguir entre los distintos tipos de excepciones.

Esta clase tiene dos constructores y dos métodos destacables:

Exception(): crea una excepción si ningún mensaje específico.

Exception(String): crea una excepción con un mensaje que detalla el tipo de excepción.

String getMessage(): el mensaje detallado, si existe (o null).

void printStackTrace(): muestra una traza que permite ver donde se generó el error (muy útil para depurar).

Tema 2. Clases y Objetos



Excepciones

Captura de excepciones

A **catch** le sigue, entre paréntesis, la declaración de una excepción. Es decir, el nombre de una clase derivada de **Exception** (o la propia **Exception**) seguido del nombre de una variable. Si se lanza una excepción que es la que deseamos capturar (o una derivada de la misma) se ejecutará el código que contiene el bloque. Así, por ejemplo:

```
catch (Exception e) { ... }
```

se ejecutará siempre que se produzca una excepción del tipo que sea, ya que todas las excepciones se derivan de **Exception**.

Tema 2. Clases y Objetos

➡ Excepciones

Captura de excepciones

Se pueden colocar varios bloques **catch**. Si es así, se comprobará, en el mismo orden en que se encuentren esos bloques **catch**, si la excepción lanzada es la que se trata en el bloque **catch**; si no, se pasa a comprobar el siguiente.

Eso sí, sólo se ejecuta un bloque **catch**. En cuanto se captura la excepción se deja de comprobar el resto de los bloques.

Veamos algunos ejemplos:

```
// Bloque 1
try {
    // Bloque 2
} catch (Exception error){
    // Bloque 3
}
// Bloque 4
```



Sin excepciones:

$1 \rightarrow 2 \rightarrow 4$

Con una excepción en el bloque 2: $1 \rightarrow 2^* \rightarrow 3 \rightarrow 4$

Con una excepción en el bloque 1: 1^*

Tema 2. Clases y Objetos

➡ Excepciones

Captura de excepciones

```
// Bloque 1
try {
    // Bloque 2
} catch (ArithmeticException ae){
    // Bloque 3
} catch (NullPointerException ne)
    // Bloque 4
}
// Bloque 5
```

Sin excepciones: $1 \rightarrow 2 \rightarrow 5$
Excepción de tipo aritmético: $1 \rightarrow 2^* \rightarrow 3 \rightarrow 5$
Acceso a un objeto nulo (null): $1 \rightarrow 2^* \rightarrow 4 \rightarrow 5$
Excepción de otro tipo diferente: $1 \rightarrow 2^*$

Tema 2. Clases y Objetos

➡ Excepciones

Captura de excepciones

```
// Bloque 1
try {
    // Bloque 2
} catch (ArithmeticException ae){
    // Bloque 3
} catch (Exception e)
    // Bloque 4
}
// Bloque 5
```

Sin excepciones:

$1 \rightarrow 2 \rightarrow 5$

➡ Excepción de tipo aritmético: $1 \rightarrow 2^* \rightarrow 3 \rightarrow 5$

Excepción de otro tipo diferente: $1 \rightarrow 2^* \rightarrow 4 \rightarrow 5$

Tema 2. Clases y Objetos

➡ Excepciones

Captura de excepciones

```
// Bloque 1
try {
    // Bloque 2
} catch (Exception e)
    // Bloque 3
} catch (ArithmeticException ae)
    // Bloque 4
}
// Bloque 5
```

¡¡OJO!!

Sin excepciones:

$1 \rightarrow 2 \rightarrow 5$

➡ Excepción de tipo aritmético: $1 \rightarrow 2^* \rightarrow 3 \rightarrow 5$

Excepción de otro tipo diferente: $1 \rightarrow 2^* \rightarrow 3 \rightarrow 5$

¡El bloque 4 nunca se ejecuta!

Tema 2. Clases y Objetos



Excepciones

Excepciones a medida o propias

El programador puede crear sus propias excepciones, si resulta que ninguna de las predefinidas es adecuada.

Se tratan igual que las predefinidas.

Para ello, se define una clase derivada de **Exception** (o de la clase deseada).

Se suele agregar un constructor con el mensaje de la excepción, que se inicializa en el constructor llamando al de la clase padre.

Además, toda excepción tiene un método **getMessage()** que devuelve un **String** con el mensaje.

Tema 2. Clases y Objetos

Declaración de las excepciones que se pueden lanzar en el método



Lanzar la excepción

Capturar la excepción



Tratar la excepción

```
package org.ip.tema02;

public class ExcepcionIntervaloPrincipal {

    public static void rango(int num, int den) throws ExcepcionIntervalo {
        if ((num > 100) || (den < -5))
            throw new ExcepcionIntervalo("Numeros fuera de rango");
    }

    public static void main(String[] args) {
        int numerador = 120;
        int denominador = 5;
        int cociente;

        try {
            rango(numerador, denominador);
            cociente = numerador / denominador;
            System.out.println("El cociente es: " + cociente);
        }

        catch (ExcepcionIntervalo ex) {
            System.out.println("Al calcular el rango se ha producido una");
            System.out.println("excepcion con el mensaje: " + ex.getMessage());
        }

        System.out.println("FIN");
    }
}
```

Ejemplo

```
package org.ip.tema02;

public class ExcepcionIntervalo extends Exception {
    public ExcepcionIntervalo (String msg) {
        super(msg);
    }
}
```

Salida

Al calcular el rango se ha producido una
excepcion con el mensaje: Numeros fuera de rango
FIN

No es necesario declarar las **RuntimeException** y derivadas

```
package org.ip.tema02;

import java.util.Scanner;

public class CocienteConMetodoYExcepcion {

    public static int cociente(int numero1, int numero2) throws ArithmeticException {
        if (numero2 == 0)
            // Se lanza la excepcion
            throw new ArithmeticException("No se puede dividir por cero");
        return numero1 / numero2;
    }

    public static void main(String[] args) {
        Scanner entrada = new Scanner(System.in);
        System.out.println("Introduzca dos enteros: ");
        int num1 = entrada.nextInt();
        int num2 = entrada.nextInt();
        try {
            int resultado = cociente(num1, num2);
            System.out.println(num1 + " / " + num2 + " es "
                + resultado);
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
            System.out.println(ex.toString());
            ex.printStackTrace();
        }
        System.out.println("La ejecución continua ...");
        System.out.println("El producto de los valores es: " + (num1 * num2));
    }
}
```

System.out.println(ex.getMessage())

System.out.println(ex.toString())

ex.printStackTrace()

Salida

```
introduzca dos enteros:
4 0
No se puede dividir por cero
java.lang.ArithmeticException: No se puede dividir por cero
java.lang.ArithmeticException: No se puede dividir por cero
    at org.ip.tema02.CocienteConMetodoYExcepcion.cociente(CocienteConMetodoYExcepcion.java:10)
    at org.ip.tema02.CocienteConMetodoYExcepcion.main(CocienteConMetodoYExcepcion.java:20)
La ejecución continua ...
El producto de los valores es: 0
```



¡MUCHAS GRACIAS!



UNIVERSIDAD DE ALMERÍA

