

## Fundamentos de Programación

---

# Paquetes y clases en Java

## Introducción

---

Uno de los principales problemas al desarrollar grandes programas es manejar la complejidad del código. La forma de afrontar dicho problema es dividir el programa en piezas modulares que tienen la función de agrupar los elementos relacionados (encapsulándolos en el mismo módulo) y separar los no relacionados (poniéndolos en módulos distintos), lo que, además, permite controlar el acceso a los diferentes componentes agrupados en un módulo, permitiendo que los demás módulos "vean" solo aquello que necesitan conocer para su funcionamiento, mientras se les oculta el resto.

## Clases

---

### Concepto y definición

En Java, el elemento modular básico es la clase. Una clase especifica un tipo de objeto en términos de las propiedades o atributos básicos que lo definen y los métodos que se pueden aplicar para modificar o calcular dichas propiedades, u otras, a partir de ellas. Por ejemplo, si quisiéramos implementar una clase para representar rectángulos, podríamos considerar que los atributos básicos de un rectángulo son la base y la altura, y dotarlo de métodos para modificar o examinar dichos atributos, así como de otros para calcular cosas como el área, el perímetro, o la longitud de la diagonal, todas las cuales se pueden calcular a partir de la base y la altura.

```
public class Rectángulo {  
    double base;  
    double altura;  
  
    public double getBase() {  
        return base;  
    }  
    public void setBase(double base) {  
        this.base = base;  
    }  
    public double getAltura() {  
        return altura;  
    }  
    public void setAltura(double altura) {  
        this.altura = altura;  
    }  
}
```

## Fundamentos de Programación

```
public double área() {
    return base * altura;
}



...

}
```

### Creación y manipulación de objetos

Una vez definida una clase, otras clases pueden declarar variables para referenciar objetos de la misma:

```
Rectángulo rec1;
Rectángulo rec2;
```

```
rec1 
rec1 
```

Las variables declaradas de esta forma referencian objetos de la clase que aparece en su declaración, pero no son el objeto propiamente, y es por lo que se conocen como *referencias*. Cuando se declaran tienen el valor *null* para indicar que no están referenciando a ningún objeto. Para crear un objeto de una clase hay que hacer algo como lo siguiente:

```
rec1 = new Rectángulo();
rec2 = new Rectángulo();
```

```
rec1  →  0.0  0.0
rec2  →  0.0  0.0
```

La expresión *new Rectángulo()* es lo que se conoce como una "llamada al constructor de la clase". Un constructor es un método especial que sirve para inicializar el estado de un objeto. Todas las clases tienen definido, automáticamente, un constructor por omisión, sin parámetros. Se caracteriza porque tiene el mismo nombre que la clase, y se distingue de los otros métodos en que no devuelve nada, ni siquiera *void*. Si se considera necesario se pueden desarrollar nuevos constructores con parámetros, además de poder hacer una definición explícita para el constructor sin parámetros. Cuando existen constructores con parámetros, el constructor sin parámetros no se puede usar, a menos que esté definido explícitamente.

```
public class Rectángulo {
    double base;
    double altura;

    public Rectángulo (double base, double altura) {
        this.base = base;
        this.altura = altura;
    }
}
```

## Fundamentos de Programación

```
public Rectángulo (Rectángulo otro) {
    this.base = otro.base;
    this.altura = otro.altura;
}

...

}
```

En el ejemplo, se han añadido a la clase *Rectángulo* dos constructores: uno para construir un objeto *Rectángulo* a partir de los datos de *base* y *altura*, y otro para construirlo a partir de otro *Rectángulo* preexistente. Estos constructores se usan igual que el constructor sin parámetros visto anteriormente, pero añadiendo los parámetros requeridos:

```
rec1 = new Rectángulo(3.2, 4.5);
rec2 = new Rectángulo(rec1);
```



Nótese el uso en la definición de los constructores de la palabra reservada *this*, que se usa para hacer referencia al objeto al que se está aplicando el método, en este caso el objeto que se está creando. La palabra *this* se utiliza, en la definición de los métodos de una clase, siempre que pueda haber una ambigüedad. Por ejemplo, en el primer constructor se usa para diferenciar cuándo se hace referencia a los campos (atributos) *base* y *altura* del objeto, de cuándo se hace referencia a los parámetros *base* y *altura* del método. En el segundo constructor, en realidad, no haría falta usar la palabra *this*, ya que *base* o *altura* por sí solas, únicamente, pueden referirse a las del objeto que se está creando, mientras que para acceder a las del objeto pasado como parámetro (*otro*, en el ejemplo) hay que prefijarlas, obligatoriamente, con el nombre del parámetro.

Una vez creado un objeto, se pueden usar los métodos ordinarios para modificarlo, examinar sus propiedades, o cualquier otra acción que dichos métodos permitan. Para llamar a uno de esos métodos, hay que prefijarlo siempre, con una expresión que referencie a un objeto de la clase (la variable *rec1* en el ejemplo siguiente). Ese objeto, al que se asocia la ejecución del método, en el código interno del método será referenciado mediante la palabra *this*, en caso necesario.

```
rec1.setBase(5.4);
double rec1Área = rec1.área();
```



Nótese que la variable *rec1Área* en el ejemplo anterior no se ha dibujado como una referencia a un objeto, a diferencia de *rec1* y *rec2*, eso se debe a que las variables de tipos primitivos (*byte*, *short*, *int*,

## Fundamentos de Programación

*long*, *float*, *double*, *boolean*, *char*) no son referencias a objetos, sino que representan directamente una entidad del tipo en cuestión.

Es importante tener en cuenta que la simple asignación, sin que intervenga un constructor o algún otro método de copia, no crea un objeto nuevo sino que cambia el objeto al que referencia la variable asignada:

```
rec2 = rec1;
```



Es lo que ocurre en el ejemplo anterior, donde *rec1* y *rec2* quedan referenciando al mismo objeto. El objeto previamente referenciado por *rec2* queda sin referencia y será destruido por el *garbage collector* de Java al ser inaccesible (no sería así, si hubiese alguna otra variable referenciándolo).

### Atributos y métodos de clase

Los atributos ordinarios son atributos de los objetos; representan información concreta sobre un objeto concreto de esa clase. Cada objeto que se crea tiene su propia copia de los atributos de objeto, con sus propios valores que puede modificar y manipular, en la medida que existan métodos para hacerlo, sin afectar a los demás objetos de la misma o de otras clases.

A veces, es necesario representar información que no pertenece a un objeto concreto de la clase, sino a la clase en su conjunto. Por ejemplo, supongamos que queremos llevar la cuenta de cuántos objetos de la clase *Rectángulo* crea un programa; esa información no tiene sentido almacenarla en un objeto concreto y, si lo hiciéramos, sería imposible mantenerla actualizada porque cada objeto tendría su propia copia y desconocería la existencia de los demás.

Ese tipo de información se guarda en un *atributo de clase*, que es un campo del que existe una única copia a la que pueden acceder todos los objetos de la clase. Un *atributo de clase* se reconoce porque su declaración va precedida por la palabra *static*.

```
public class Rectángulo {
    ➔ static int creados = 0;
    double base;
    double altura;
    ...
}
```

## Fundamentos de Programación

En este ejemplo, para conseguir el efecto que se buscaba, basta con que cada constructor incremente la variable *creados*.

```
public Rectángulo (double base, double altura) {  
    Rectángulo.creados++;  
    this.base = base;  
    this.altura = altura;  
}
```

Nótese que, en el ejemplo, el campo *creados* se llama prefijándolo con el nombre de la clase, no con la palabra *this*; la palabra *this* hace referencia a un objeto concreto, pero *creados* no pertenece a ningún objeto concreto, sino a la clase en su conjunto. También es cierto que, en el ejemplo, no es necesario, en realidad, hacer esa prefijación puesto que, en ese contexto concreto, *creados* no se confunde con nada.

Si ahora quisiéramos tener un método que devolviera el valor de *creados*, se podría hacer como un método ordinario que deba asociarse a un objeto para ejecutarse, ya que todos los objetos tienen acceso a los campos de clase, pero sería más lógico definirlo como método de clase, lo que se hace, igual que con los atributos, usando la palabra *static*.

```
public static int getCreados() {  
    return Rectángulo.creados;  
}
```

Para llamar a este método desde fuera de la clase sí que sería necesario prefijarlo con el nombre de la clase a la que pertenece:

```
System.out.println(Rectángulo.getCreados());
```

Un método de clase, como *getCreados*, solo puede acceder a atributos de clase ya que su llamada no se asocia a un objeto concreto, a cuyos atributos pudiera acceder.

## Fundamentos de Programación

### Paquetes y visibilidad

#### Paquetes

Las clases que tienen algún tipo de relación entre sí se agrupan en paquetes, proporcionando un nivel adicional de organización del código. Indicar que una clase pertenece a un determinado paquete es bastante sencillo, basta con incluir una línea al principio del archivo `.java` donde se define la clase:

ARCHIVO: Rectángulo.java

```
package es.ulpgc.eii.figuras;
```



```
public class Rectángulo {  
    ...  
}
```

Todas las clases de un paquete "ven" a todas las demás clases que pertenecen al mismo paquete y pueden declarar variables que referencien a objetos de cualquiera de ellas; otra cosa es qué uso le puedan dar. Las clases que pertenecen a otro paquete, sin embargo, solo son "visibles" si han sido declaradas usando el modificador *public*, como la del ejemplo. Aún así, para poder usarlas, es necesario incluir, antes de la cabecera de la clase que quiere hacerlo, una cláusula *import* que lo diga.

ARCHIVO: MyClass.java

```
package es.ulpgc.eii.otro;
```

```
import es.ulpgc.eii.figuras.Rectángulo;
```



```
public class MyClass {  
    ...  
}
```

Nótese que el nombre completo de una clase incluye el nombre del paquete al que pertenece (en el ejemplo, el nombre completo de la clase *Rectángulo* es *es.ulpgc.eii.figuras.Rectángulo*), aunque, generalmente, no es necesario usar dicho nombre completo.

Cuando en el archivo de la definición de una clase se omite el nombre del paquete al que pertenece, esa clase pasa a formar parte del paquete *default*. Es recomendable asignar las clases a paquetes concretos y evitar, en la medida de lo posible, dejarlas en el paquete *default*, ya que las clases que pertenecen a este paquete no pueden ser usadas mas que por las clases del propio paquete *default*.

## Fundamentos de Programación

### Modificadores de visibilidad

La palabra *public*, que aparece en los ejemplos anteriores en la cabecera de la clase, actúa como un modificador de visibilidad. Los modificadores de visibilidad pueden aplicarse a nivel de clase, para indicar qué clases de un paquete pueden o no "verse" desde fuera, o a nivel de componente, para indicar a qué componentes de una clase tienen acceso otras clases, y qué tipo de acceso tienen.

A las clases se les pueden aplicar el atributo *public* o ninguno. El atributo *public* significa que la clase es visible fuera del paquete, mientras que la ausencia de modificadores significa que no lo es.

A los componentes de una clase se les pueden aplicar los modificadores: *public*, *private*, *protected*, o ninguno. La ausencia de modificadores indica que el componente es visible para todas las clases del mismo paquete, pero no para clases definidas en otros paquetes. El modificador *public* indica que el componente es visible para cualquier clase de cualquier paquete. El modificador *private* indica que el componente solo es visible dentro de su propia clase. Por último, *protected* hace que el componente sea visible para cualquier subclase<sup>1</sup> derivada de la clase a la que pertenece el componente. La siguiente tabla resume el significado de los modificadores de visibilidad aplicados a los componentes de una clase:

	La propia clase	Otras clases en el mismo paquete	Subclases	Clases en otros paquetes
<b>public</b>	X	X	X	X
<b>protected</b>	X	X	X	
<b>Ninguno</b>	X	X		
<b>private</b>	X			

Tabla 1

A la hora de poner modificadores de visibilidad se debe tender a la paranoia, no revelando más que lo estrictamente necesario; es decir, sólo declararemos *public* aquello que no quede más remedio que declarar *public* y declararemos *private* todo lo que podamos declarar *private*. Por ejemplo, en la clase *Rectángulo*, los atributos deberían llevar el modificador *private*, en vez de ninguno.

```
public class Rectángulo {
    private static int creados = 0;
    private double base;
    private double altura;
    ...
}
```

<sup>1</sup> ¡Que no cunda el pánico! En breve veremos qué son las subclases

## Fundamentos de Programación

### Jerarquía de clases en Java

Java, como lenguaje orientado a objetos, permite formar jerarquías de clases que derivan unas de otras, usando un mecanismo conocido como herencia que, en conjunción con otro mecanismo conocido como polimorfismo, confiere una gran potencia a este tipo de lenguajes. La herencia y el polimorfismo dan lugar a profundas implicaciones en el diseño y funcionamiento de los programas, implicaciones que escapan a nuestros objetivos en este momento. Sin embargo, creemos conveniente ofrecer un par de pinceladas sobre los conceptos de subclase y superclase, la jerarquía de clases de Java, y otros aspectos relacionados.

### Subclases

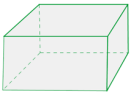
Para maximizar la reutilización del código<sup>2</sup>, una clase puede definirse, como extensión de otra preexistente, ampliándola o especializándola:

```
package es.ulpgc.eii.figuras;

public class Ortoedro3 extends Rectángulo
{
    private double grosor;

    public Ortoedro(double base,
                    double altura,
                    double grosor) {
        super(base, altura);
        this.grosor = grosor;
    }

    public double Volumen() {
        return this.área() * this.grosor;
    }
}
```



```
package es.ulpgc.eii.figuras;

public class Cuadrado extends Rectángulo
{
    public Cuadrado(double lado) {
        super(lado, lado);
    }

    public void setBase(double base) {
        this.base = base;
        this.altura = base;
    }

    public void setAltura(double altura) {
        this.altura = altura;
        this.base = altura;
    }
}
```

Un objeto de una subclase es, al mismo tiempo y a todos los efectos, un objeto de la superclase con atributos, constructores y métodos adicionales. Dentro de la subclase se puede hacer referencia a la superclase usando la palabra *super*.

Normalmente, en los constructores de una subclase la primera instrucción será una llamada a un constructor de la superclase, al que se le pasan los parámetros necesarios para construir un objeto de la clase base sobre el que, después, se terminará de construir el objeto de la subclase, como se puede

<sup>2</sup> Entre otras razones...

<sup>3</sup> Se está usando una definición "sui géneris" de ortoedro como "rectángulo grueso", lo cual quizá no sea lo más adecuado desde el punto de vista de la geometría, pero resulta útil como ejemplo de clase que extiende a otra



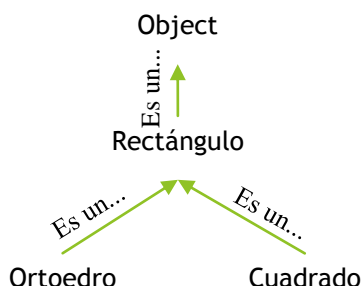
## Fundamentos de Programación

ver en el ejemplo con la clase *Ortoedro* (el constructor de la clase *Cuadrado* no necesita hacer nada más, una vez que se ha construido un "Rectángulo con dos lados iguales").

Si, como pasa en el ejemplo con la clase *Cuadrado*, se añade un método con la misma signatura (parámetros y tipo devuelto) existente en la superclase, lo oculta, de manera que, para los objetos de la subclase se utilizarán las versiones definidas en la subclase.

### Jerarquía de clases

En Java las clases forman una jerarquía que deriva de una superclase llamada *Object*. Todas las clases derivan directa o indirectamente de la clase *Object* ( $\Rightarrow$  cualquier objeto, de cualquier clase, es un *Object*). Una clase que en su definición no declare extender a otra clase, como la clase *Rectángulo* que hemos usado en los ejemplos, es una subclase directa de la clase *Object*. Una clase que en su definición declare extender a otra clase, como las clases *Ortoedro* y *Cuadrado*, es una subclase indirecta de la clase *Object*, a través de sus superclases.



El que todos los objetos, de cualquier clase, sean objetos de la clase *Object*, hace que dispongan de una serie de métodos comunes que todos pueden utilizar, aunque, generalmente, cada clase deberá redefinir algunos de ellos, ocultando los heredados, para que funcione correctamente de acuerdo a sus características particulares. A continuación se describen algunos de estos métodos comunes:

Modificador y tipo	Método	Descripción
<b>protected</b> Object	<code>clone()</code>	Crea y devuelve una copia del objeto referenciado por <i>this</i>
<b>boolean</b>	<code>equals(Object obj)</code>	Indica si <i>obj</i> es igual al objeto <i>this</i>
String	<code>toString()</code>	Devuelve una representación en forma de ristra del objeto referenciado por <i>this</i>

## Fundamentos de Programación

---

### Tipos primitivos y *wrappers*

Los tipos primitivos (*byte*, *short*, *int*, *long*, *float*, *double*, *boolean* y *char*) no se integran en la jerarquía de la clase *Object* de Java. No obstante, por cada tipo primitivo, Java ofrece también una clase "*wrapper*", (*Byte*, *Short*, *Integer*, *Long*, *Float*, *Double*, *Boolean* y *Character*) que representa lo mismo que el tipo primitivo, pero integrado en la jerarquía de clases. Los *wrapper* se usan para encapsular automáticamente a los valores de los tipos primitivos cuando una operación necesite tratar datos de un tipo primitivo como un *Object*. Como se ha indicado, el proceso de conversión entre un tipo primitivo y su *wrapper*, y viceversa, se desencadena automáticamente cuando hace falta y resulta transparente al programador.

Una aportación adicional de los *wrapper* es proporcionar un número importante de métodos para manipular valores de ese tipo; por ejemplo, la clase *Character* proporciona métodos para saber si un carácter es mayúscula o minúscula, si es un dígito, un separador, una letra, o a qué otra categoría puede pertenecer, etc.