

# UNIDAD 7: PROGRAMACIÓN ORIENTADA A OBJETOS

**Profesor:** José Ramón Simó Martínez

## Contenido

---

<b>1. Introducción.....</b>	<b>2</b>
1.2. Definición de POO .....	2
<b>2. Abstracción .....</b>	<b>3</b>
<b>3. Clases y objetos.....</b>	<b>3</b>
3.1. Definición de clases .....	4
3.2. Instanciación de objetos .....	4
<b>4. Atributos y métodos.....</b>	<b>5</b>
4.1. Definición y acceso a atributos .....	5
4.2. Definición y acceso a métodos .....	6
4.3. Constructores .....	8
<b>5. Encapsulamiento.....</b>	<b>10</b>
5.1. Modificadores de acceso.....	11
5.2. Getters y setters .....	13
<b>6. Empaquetado de clases .....</b>	<b>14</b>
6.1. Paquetes integrados .....	15
6.2. Paquetes definidos por el desarrollador .....	15
6.3. Ejemplo de creación y uso de paquetes propios.....	18
<b>7. Bibliografía.....</b>	<b>19</b>

## 1. Introducción

En este punto del curso y con todo lo que has aprendido ya eres capaz de resolver gran parte de los problemas, a nivel de programación, utilizando las estructuras de control, arrays y métodos. Sin embargo, los proyectos empiezan a ser más complejos, incluyendo por ejemplo interfaces gráficas, bases de datos o software de gran escalabilidad. Esto requiere de un nivel mayor de abstracción en el diseño del software, así como un lenguaje que permita organizar y aplicar ese nivel de abstracción.

En esta unidad introduciremos la programación orientada a objetos (POO) y su aplicación en el lenguaje de programación Java, con el objetivo de aprender a desarrollar proyectos de software de mayor calidad, robustez, eficiencia y seguridad, entre otros.

Al terminar esta unidad deberás ser capaz de:

- Comprender las bases de la POO.
- Entender el concepto de objeto y clase dentro de la POO.
- Definir y utilizar tus propias clases.
- Definir los atributos y métodos de una clase.
- Crear y utilizar constructores de clase.
- Desarrollar programas sencillos aplicando el paradigma orientado a objetos básico en lenguaje Java.

### 1.2. Definición de POO

La POO es un estilo de programación donde todos los elementos que forman parte del problema se conciben como objetos, definiendo cuáles son sus atributos y comportamiento, cómo se relacionan entre sí y cómo están organizados.

Los principales elementos que componen la POO:

- Abstracción
- Clases y objetos
- Encapsulamiento
- Herencia
- Polimorfismo

De forma teórica también forman parte el **Principio de Ocultación de Información** y la **Modularidad**. No entraremos en definiciones sobre estos conceptos, pero si quieres saber más te dejo este [enlace](#).

En esta unidad estudiaremos los tres primeros elementos. Dejaremos la **Herencia** y **Polimorfismo** para la siguiente unidad como conceptos avanzados de POO.

## 2. Abstracción

En el pensamiento humano tendemos a abstraer el mundo real con el objetivo de simplificar la realidad. De un objeto, problema, situación, etc., obtenemos la información esencial y deseamos aquella que no sirve para nuestro objetivo. Por ejemplo:

No necesitamos saber la física que hay detrás de un **microondas** para poder calentar nuestra comida. Solamente debemos saber:

- Las características del microondas (tamaño, potencia, etc.).
- Las funciones que nos ofrece (regular tiempo y potencia, descongelar, función grill, etc.).

A estas características y funciones las denominamos en los lenguajes de POO como Atributos y Métodos, respectivamente.

### Nota

A estas características y funciones las denominamos en los lenguajes de POO como Atributos y Métodos, respectivamente.

### La abstracción como técnica de programación

Concluimos que la programación es una tarea compleja y mediante la **abstracción** es posible elaborar software que permita solucionar problemas cada vez más grandes.

### Nota

En el módulo de Entornos de Desarrollo (ED) se estudia con más detalle los conceptos teórico-prácticos del paradigma orientado a objeto, como por ejemplo los diagramas UML. En esta unidad se dará por entendido que el estudiante ha adquirido dichos conocimientos del módulo de ED.

## 3. Clases y objetos

En los lenguajes de programación un objeto está compuesto por:

- **Atributos:** definen el **estado** del objeto.
- **Métodos:** definen el **comportamiento** del objeto.

Una vez entendido el concepto de objeto podemos definir qué es una clase en los lenguajes de programación:

*“Podemos entender como clase el molde a partir del cual se crearán los objetos de dicha clase.”*

Una analogía para entender mejor la diferencia entre objeto y clase sería el diccionario, donde la clase sería la palabra y su definición, y los objetos cada elemento real de dicha palabra. Por ejemplo, en el diccionario podemos encontrar la definición de Persona (la clase) y en el mundo real tenemos miles de personas (objetos).

Una clase, al igual que los objetos, se compone de:

- **Información:** **campos** (atributos, propiedades)
- **Comportamiento:** **métodos** (operaciones, funciones)

Diremos que un objeto es una **instancia** de una clase. Por ejemplo:

Clase	Objetos (Instancias)
Persona	<i>Ramon, Gala, Jose, Reme, Melis...</i>
Coche	<i>KITT, DeLorean DMC-12, Batmovil...</i>
Jedi	<i>Luke Skywalker, Yoda, Obi-Wan Kenobi, Ansoka Tano...</i>
Nave	<i>Halcón milenario, USS Enterprise, USSCS Prometheus...</i>
Planeta	<i>Tierra, Júpiter, Arrakis, Tatooine, Krypton...</i>
String	<i>"Hola mundo"...</i>
Date	<i>17-01-2023...</i>

A continuación, vamos a ver como se definen las clases e instancian los objetos en **Java**.

### 3.1. Definición de clases

El esquema de una clase en Java es la siguiente:

```
class NombreDeClase {

    // Atributos

    // Constructores

    // Métodos

}
```

### 3.2. Instanciación de objetos

A lo largo del curso ya hemos utilizado algunas de las clases predefinidas de Java, como por ejemplo la clase **String**, **Random**, **Date**, etc. Para poder utilizar estas clases hacíamos lo siguiente, por ejemplo:

```
Random r = new Random(); // Instanciamos un objeto de la clase Random
Date fecha = new Date(); // Instanciamos un objeto de la clase Date
...
```

Ahora ya sabemos que lo que estábamos haciendo era instanciar un objeto de una clase. Hay dos elementos que participan en la instancia de un objeto:

- La palabra clave **new**.
- El **constructor de clase**, que debe coincidir exactamente con el nombre de la clase.

El esquema general para instanciar (crear) un objeto de una clase será:

```
NombreDeClase objeto = new NombreDeClase();
```

Más adelante estudiaremos las particularidades a la hora de instanciar objetos utilizando los **constructores de clase con parámetros**.

## 4. Atributos y métodos

A continuación, vamos a estudiar cómo se definen en Java los atributos y métodos de una clase, así como su uso por parte de un objeto.

### 4.1. Definición y acceso a atributos

Definen las propiedades del objeto de la clase. Para ello, los atributos son las variables (tipos de datos primitivos u objetos) que ya conocemos.

Por ejemplo:

Clase	Atributos
Persona	<i>nombre, edad, teléfono,...</i>
Coche	<i>nº de bastidor, tipo, marca, color,...</i>
Jedi	<i>nivel de fuerza, experiencia, categoría, lado,...</i>
Nave	<i>tamaño, potencia, capacidad,...</i>

Existen dos tipos de atributos: de instancia y de clase.

- **De instancia:** hay una copia de un campo de instancia por cada objeto de la clase.
- **De clase (*static*):** hay una única copia de un campo *static* en el sistema. Es el equivalente a una variable global en otros lenguajes de programación. El campo *static* es accesible a través de la propia clase (sin necesidad de instanciar la clase).

Por ejemplo:

Clase	Atributos de instancia	Atributos de clase
Persona	<i>nombre, edad, teléfono,...</i>	<i>Nº de personas creadas</i>

Un ejemplo de clase Persona con sus atributos definidos:

```
class Persona {
    String nombre;
    int edad;
    String teléfono;
    static int numeroDePersonasCreadas;
}
```

Para acceder a los atributos de una instancia se utiliza la **sintaxis del objeto**:

```
Persona p1 = new Persona();
p1.nombre = "Leia";
p1.edad = 38;
```

Acceso a las variables static se utiliza la **sintaxis de clase**:

```
Persona.numeroDePersonasCreadas++;
System.out.println("Persona.numerosDePersonasCreadas");
```

## 4.2. Definición y acceso a métodos

En anteriores unidades ya estudiamos el concepto de función como estructura que permite modular un programa. En Java a las funciones se les conoce como **métodos**. Por tanto, ya conocemos la estructura básica de un método: recibe parámetros (o no), ejecuta instrucciones y devuelve (o no) un dato.

No obstante, al igual que en los atributos existen dos tipos de métodos en la POO: de instancia y de clase.

- De instancia: se invoca sobre un objeto de la clase, al cual tiene acceso mediante la palabra reservada **this**. Sus variables de instancia son accesibles de manera directa.
- De clase (**static**): no opera sobre un objeto de la clase, y la palabra reservada **this** no es válida en su interior.

### Nota

En los métodos de clase NO podemos utilizar variables de instancia.

Por ejemplo:

Clase	Métodos de instancia	Métodos de clase
Persona	<i>esMayorDeEdad()</i>	<i>obtenerPoblacion(), incrementarPoblacion()</i>

En código Java:

```
class Persona {
    // Atributos
    String nombre;
    int edad;
    String teléfono;
    static int numeroDePersonasCreadas;

    // Métodos
    void esMayorDeEdad() {
        if (edad >= 18) // atributo de instancia
            System.out.println(nombre + " es mayor de edad");
        else
            System.out.println(nombre + " no es mayor de edad");
    }
}
```

```
static int obtenerPoblacion() {  
    return numeroDePersonasCreadas; // atributo de clase  
}  
  
static void incrementarPoblacion() {  
    numeroDePersonasCreadas++; // atributo de clase  
}  
}
```

#### 4.2.1. La palabra reservada *this*

La palabra reservada **this** permite que se pueda hacer referencia al objeto actual de la clase. De este modo, nos servirá para hacer referencia a los propios atributos y métodos de la clase. Su sintaxis es *this.nombreDeVariable*.

Por ejemplo:

```
void esMayorDeEdad() {  
    if (this.edad >= 18) // atributo de instancia  
        System.out.println(nombre + " es mayor de edad");  
    else  
        System.out.println(nombre + " no es mayor de edad");  
}
```

En el ejemplo anterior podemos obviar el uso del **this**, ya que Java detecta que *edad* es un atributo de la clase. No obstante, como veremos más adelante, el uso del *this* será necesario en muchos casos dentro de los métodos de la clase.

#### Nota

Dependiendo de la guía de estilo del proyecto en el que se trabaje se recomendará usar o no el *this* siempre que se haga referencia a un atributo o método de la clase.

#### 4.2.2. Acceso a los métodos

Para acceder a los métodos de una instancia se utiliza la **sintaxis del objeto** (objeto.metodo):

```
Persona p1 = new Persona(); // p1 es un objeto de la clase persona  
p1.esMayorDeEdad(); // método de instancia
```

Acceso a los métodos de clase (*static*) se utiliza la **sintaxis de clase** (Clase.metodo):

```
int poblacionTotal = Persona.obtenerPoblacion(); // método de clase
```

### 4.2.3. Sobrecarga de métodos

Los métodos de una clase pueden tener el mismo nombre, pero diferentes parámetros. Cuando se invoca un método el compilador compara el número y tipo de los parámetros y determina qué método debe invocar.

Ejemplo:

```
class Persona {  
    double ahorros;  
  
    void ingresarDinero(double cantidad) {  
        this.ahorros += cantidad;  
    }  
  
    void ingresarDinero(double cantidad, String moneda) {  
        // Procesar el ingreso  
    }  
}
```

Ahora podemos utilizar el mismo nombre de método, pero pasando diferentes parámetros:

```
Persona p1 = new Persona();  
p1.ingresarDinero(128.25);  
p1.ingresarDinero(50, "€");
```

En resumen, esta técnica se conoce como **sobrecarga de métodos** y nos permite definir más de un método con el mismo nombre con la condición de que sus parámetros sean distintos

#### Nota

Si no existiera la sobrecarga de métodos deberíamos de haber nombrado a los anteriores métodos del ejemplo como “ingresarDinero1(double cantidad)”, “ingresarDinero2(double cantidad, String moneda)”. La ventaja de esta técnica práctica es reducir el número de nombres de método distintos y agrupar características comunes.

### 4.3. Constructores

Un constructor es un método especial invocado para instanciar e inicializar un objeto de la clase, y se invoca con la sentencia **new**.

Los métodos constructores cumplen con estas características:

- ✓ Tienen el mismo que la clase.
- ✓ Pueden tener cero (conocido como **constructor por defecto**) o más parámetros (conocido como **constructor con parámetros**).
- ✓ Deben declararse públicos (**public**) para que se puedan invocar fuera de la clase, de otra forma se restringiría el acceso a la creación de objetos.
- ✓ No tiene tipo de retorno, ni siquiera **void**.



- ✓ Sólo se ejecuta cuando es invocado para crear una instancia de un objeto.
- ✓ Si la clase no tiene ningún constructor, el sistema crea un constructor por defecto.
- ✓ Si la clase tiene algún constructor, debe usarse alguno de los constructores definidos al instanciar la clase (en este caso el sistema no crea un constructor por defecto).

#### 4.3.1. Constructores por defecto

El objetivo de crear constructores por defecto es inicializar el valor de los atributos. Debemos hacernos la siguiente pregunta: ¿qué valor queremos que tengan los atributos de un objeto al crear dicho objeto?

En la mayoría de los casos los valores iniciales son los cero para valores numéricos, cadena vacía para String, etc. Sin embargo, en algunos casos queremos que tengan un valor concreto. Por ejemplo, en una clase llamada *Tripode* queremos que su atributo *numeroDePatatas* (de tipo entero) sea inicialmente 3.

Ejemplo de constructor por defecto en la clase *Persona*:

```
class Persona {  
    // Atributos  
    String nombre;  
    int edad;  
    String teléfono;  
    static int numeroDePersonasCreadas;  
  
    // Constructor por defecto  
    public Persona() {  
        this.nombre = "";  
        this.edad = 0;  
        this.teléfono = "";  
        numeroDePersonasCreadas = 0; // tipos static se usan aquí  
    }  
}
```

##### Nota

Observa en ejemplo anterior que no usamos *this* en el atributo de tipo *static*. Esto es porque *this* sólo se usa en atributos de instancia.

En caso de no crear el constructor por defecto, Java crea uno internamente e inicializa las variables a un valor inicial. No entraremos en detalle por ahora sobre este aspecto, así que nosotros crearemos siempre el constructor por defecto e inicializaremos los atributos de la clase.

#### 4.3.2. Constructores con parámetros

La sobrecarga de métodos también se aplica a los constructores de la clase. Es decir, podemos tener constructores con el mismo nombre, pero diferentes parámetros. Esto permitirá que podamos tener más de un constructor de clase según nuestras necesidades de diseño a la hora de instanciar objetos de la clase.

Por ejemplo:

```
class Persona {  
    // Atributos  
    String nombre;  
    int edad;  
    String teléfono;  
    static int numeroDePersonasCreadas;  
  
    // Constructor por defecto  
    public Persona() {  
        this.nombre = "";  
        this.edad = 0;  
        this.teléfono = "";  
        numeroDePersonasCreadas = 0; // tipos static se usan aquí  
    }  
  
    // Constructores con parámetros  
    public Persona(String nombre) {  
        this.nombre = nombre;  
    }  
  
    public Persona(String nombre, int edad) {  
        this.nombre = nombre; // Uso necesario del this!!  
        this.edad = edad;  
    }  
}
```

#### Nota

Como puedes observar el uso del **this** es necesario en este caso para diferenciar el nombre del parámetro de entrada y el atributo de la clase, cuando estos por claridad de código se nombran igual.

Para instanciar un objeto usando los constructores de parámetros se haría así:

```
Persona p1 = new Persona();  
  
Persona p2 = new Persona("Adrián");  
  
Persona p3 = new Persona("Reme", 62);
```

En el ejemplo se han instanciado tres objetos de la clase Persona utilizando diferentes los constructores de clase que hemos definido anteriormente.

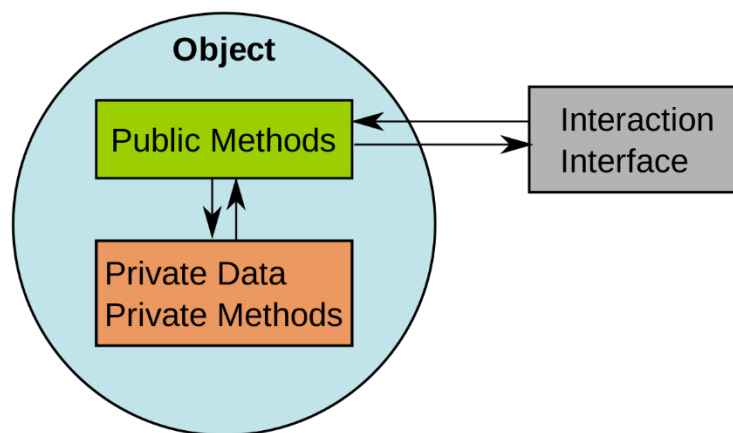
## 5. Encapsulamiento

Uno de los objetos de la POO es agrupar tanto los datos como las funciones dentro de una estructura que llamamos Clase. Por otra parte, en el ecosistema de un proyecto software orientado a objetos conviven diferentes clases y objetos, que de una manera u otra se pueden relacionar.

Surge por tanto la necesidad de crear un mecanismo que permita agrupar los datos en una unidad llamada Clase, que permita indicar cuáles de sus atributos y métodos se pueden compartir, y además qué clases pueden acceder o no a ellos. Este mecanismo se denomina *Encapsulamiento*, que en resumen permite limitar el acceso a los elementos de la propia clase.

El encapsulamiento es consecuencia del concepto de *Principio de Ocultación de la Información* que indica la necesidad de ocultar la representación interna de un objeto o su estado desde el exterior.

En la siguiente imagen podemos observar cómo un objeto incluye diferentes métodos y atributos, pero solamente aquellos que son públicos podrán interactuar con el exterior de la clase. A estos métodos públicos también los denominamos *Interfaces* de la clase.



A continuación, veremos los diferentes modificadores de acceso que definen la visibilidad de los métodos y atributos de una clase.

### 5.1. Modificadores de acceso

Durante el curso ya has utilizado el modificador `public` en el código, aunque lo haya generado el IDE:

```
public class Principal {
    public static void main(String[] args) {
        // Código...
    }
}
```

La palabra reservada **public** es un modificador de acceso, que define el nivel de acceso para las clases, atributos, métodos y constructores.

Los **modificadores de acceso** de Java que modifican el acceso a las clases son:

- **public:** la clase es accesible por cualquier otra clase.
- **default:** la clase solo es accesible por otras clases del mismo paquete. El modificador default es aplicado cuando no se especifica ningún modificador.

Los modificadores de acceso de Java que modifican el acceso a los atributos, métodos y constructores son:

- **public:** es accesible por todas las clases.
- **private:** es accesible solamente dentro de la clase donde está declarado.
- **default:** es accesible solamente dentro del mismo paquete. El modificador default es aplicado cuando no se especifica ningún modificador.
- **protected:** es accesible en el mismo paquete y las subclases.

#### Nota

Los paquetes los veremos en los siguientes apartados mientras que concepto de subclases lo introduciremos en la siguiente unidad.

Una tabla de resumen de la accesibilidad de los modificadores de acceso:

Modificador	Clase	Paquete	Subclase	Otros
public	Sí	Sí	Sí	Sí
protected	Sí	Sí	Sí	No
(default)	Sí	Sí	No	No
private	Sí	No	No	No

Ejemplo:

```
public class Persona {
    // Atributos
    private String nombre;
    public int edad; // no es nada recomendable hacer public el atributo!!
    protected String telefono; // lo estudiaremos en subclases
    static int numeroDePersonasCreadas; // default

    // Métodos
    public void esMayorDeEdad() {
        if (edad >= 18) // atributo de instancia
            System.out.println(nombre + " es mayor de edad");
        else
            System.out.println(nombre + " no es mayor de edad");
    }

    static int obtenerPoblacion() {
        return numeroDePersonasCreadas; // atributo de clase
    }

    static void incrementarPoblacion() {
        numeroDePersonasCreadas++; // atributo de clase
    }
}
```

Ejemplo de uso en otra clase Principal que esté en el mismo paquete que la clase Persona:

```
public class Principal {  
  
    public static void main(String[] args) {  
        Persona p1 = new Persona();  
  
        // Dará error al ser private  
        String miNombre = p1.nombre;  
  
        // Funcionará, pero no es para nada adecuado acceder de esta forma a los  
        // datos de la clase  
        int miEdad = p1.edad;  
    }  
}
```

#### Nota

En el ejemplo se indican los atributos con diferentes tipos de modificadores. Como veremos en el siguiente apartado no es recomendable establecer los atributos como public o default (sin definir).

## 5.2. Getters y setters

Como consecuencia de la Encapsulación y la ocultación de los datos sensibles al usuario, una buena práctica de programación es:

1. Declarar todos los atributos de la clase como *private*.
2. Ofrecer métodos declarados como *public* para poder acceder y actualizar los valores de esos atributos *private*.

Estos métodos se conocen como getters y setters:

- **getter:** método que devolverá el valor de un atributo private de la clase.
- **setter:** método que almacena o actualiza un valor de un atributo private de la clase.

La sintaxis para ambos métodos es escribir get o set, seguido del nombre del atributo con el primer carácter en mayúsculas. Por ejemplo:

```
class Persona {  
    // Atributos  
    private String nombre;  
    private int edad;  
    private String teléfono;  
    private static int numeroDePersonasCreadas;  
  
    // Constructores...  
    // Getter  
    public int getEdad() {  
        return edad; // o también return this.edad;  
    }  
}
```

```
// Setter
public void setEdad(int edad) {
    this.edad = edad;
}
}
```

A continuación, un ejemplo para probar los getters y setters:

```
public class Principal {
    public static void main(String[] args) {
        Persona p1 = new Persona();
        p1.setEdad(40);
        int edadPersona = p1.getEdad(); // devolverá 40
        System.out.println(edadPersona); // mostrará 40
    }
}
```

## 6. Empaquetado de clases

### Nota

Durante el curso has utilizado necesariamente esta herramienta de Java para solucionar los ejercicios propuestos. Quizás lo hacías de forma inconsciente ya que el entorno de desarrollo (en este curso Eclipse) suele advertir o directamente añade los paquetes necesarios conforme los vamos escribiendo en el programa. La motivación de este apartado es entender mejor el concepto de paquete, pero sobre todo aprender a crear tus propios paquetes en Java.

La encapsulación de la información dentro de las clases ha permitido llevar a cabo el proceso de ocultación, que es fundamental para el trabajo con clases y objetos. Cuando un proyecto empieza a crecer en cuanto al número de clases y a las relaciones que se establecen entre ellas por el modelo de datos diseñado, quizás necesites organizar dichas clases en un nivel superior de encapsulamiento conocido como **empaquetado** (**package** en Java).

Un **paquete** en Java se utiliza para agrupar clases relacionadas entre sí bajo un mismo nombre. Es aconsejable reunir aquellas clases que tienen unas características en común. Por ejemplo, durante el curso ya hemos utilizado el paquete **java.util**, en el que podemos encontrar todas las clases fundamentales (útiles) que los desarrolladores del lenguaje Java han creído conveniente agrupar (Scanner, Date, Calendar, Random, etc).

Como consejo, puedes pensar que los paquetes son carpetas de un directorio de archivos. Se usan paquetes para evitar conflictos y facilitar el mantenimiento del código. Los paquetes se dividen en dos categorías:

- **Paquetes integrados** (built-in Packages): paquetes de la [API](#) de Java.
- **Paquetes definidos por el usuario**: Java permite que los desarrolladores puedan crear sus propios paquetes.

### 6.1. Paquetes integrados

Para acceder a las clases de un paquete se debe utilizar la palabra clave `import` seguida de una estructura organizativa separada por puntos:

```
import paquete.nombrepaquete.nombreclase;
```

Para utilizar todas las clases del paquete sustituimos la Clase por un asterisco (\*):

```
import paquete.nombrepaquete.*;
```

#### Nota

Es recomendable que importemos **únicamente** las clases que vayamos a utilizar para nuestro proyecto. Dicho de otra forma, evitar en la medida de lo posible la tentación de usar el asterisco (\*).

Ejemplo en Java:

```
import java.util.Scanner;

class TestScanner {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Introduce tu nombre: ");
        String nombre = sc.nextLine();
    }
}
```

### 6.2. Paquetes definidos por el desarrollador

Para crear tus propios paquetes necesitas saber que Java utiliza un sistema de directorios para guardarlos. Como habíamos dicho al principio de este apartado, los paquetes son carpetas desde el punto de vista del sistema operativo. En el siguiente ejemplo vemos el árbol de directorios de un paquete en Java en el path C:\MiProyectoJava\src\nombrePaquete\MiClase.java:

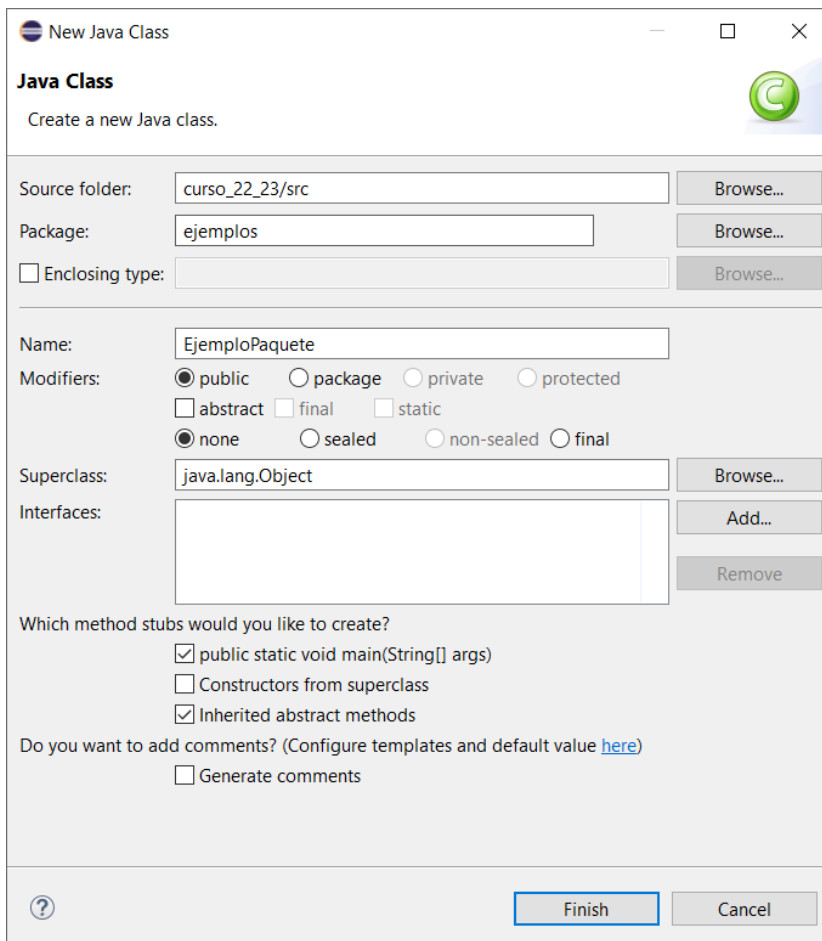
```
|__ MiProyectoJava
    |__ src
        |__ nombrePaquete
            |__ MiClase.java
```

Para crear un paquete debemos usar la palabra reservada `package` seguida del nombre del paquete. Esta instrucción se debe situar al inicio del fichero .java. Por ejemplo:

```
package nombrePaquete;  
  
class MiClase {  
    public static void main(String[] args) {  
        System.out.println("Hola mundo");  
    }  
}
```

Sin embargo, en este curso estamos utilizando un entorno de desarrollo que permite crear el paquete desde una ventana gráfica. A continuación, se mostrarán los pasos para crear un paquete en el IDE Eclipse:

1. Dentro de tu proyecto realiza los pasos abrir la ventana de creación de Clases. Como se ve en el ejemplo, escribe el nombre del paquete en el segundo campo donde pone **Package**. Sigue los pasos para rellenar el nombre de clase y demás opciones:



The screenshot shows the 'New Java Class' dialog box in Eclipse. The 'Package' field is filled with 'ejemplos'. The 'Name' field is filled with 'EjemploPaquete'. The 'Modifiers' section has 'public' selected. The 'Superclass' field is filled with 'java.lang.Object'. The 'Which method stubs would you like to create?' section has 'public static void main(String[] args)' and 'Inherited abstract methods' checked. The 'Do you want to add comments?' section has 'Generate comments' unchecked. The 'Finish' button is highlighted.

2. Una vez le das al botón **Finish**, verás que te crea una clase con la instrucción situada arriba para crear paquetes:



```
EjemploPaquete.java ×
1 package ejemplos;
2
3 public class EjemploPaquete {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7         System.out.println("Ejemplo de paquete de usuario");
8     }
9
10 }
```

Ahora podrás ver en la vista de Package Explorer que Eclipse ha creado tanto el paquete y la clase:

```
Package Explorer ×
└─ curso_22_23
   └─ JRE System Library [JavaSE-17]
      └─ src
         └─ ejemplos
            └─ EjemploPaquete.java
```

Si nos vamos al sistema de directorios del sistema operativo (Windows en este caso) verás que el paquete es realmente un directorio:

```
← → ▾ ↑ > Este equipo > Windows (C:) > Curso-daw > Curso_22_23 > src > ejemplos
```

Nombre
EjemploPaquete

```
└─ Curso-daw
   └─ .metadata
      └─ Curso_22_23
         └─ .settings
            └─ bin
               └─ src
                  └─ ejemplos
```

Por otra parte, cuando separamos por puntos (.) en el **package** lo que estamos haciendo es crear cada vez un subdirectorio del directorio anterior. Por ejemplo:

```
package ejemplos.faciles;
```

En este caso, el sistema operativo crear el directorio *ejemplos* y dentro de este el subdirectorio *faciles*.

#### Nota

En eclipse, cuando quieres añadir algo al proyecto con *New...* verás que en el desplegable también tienes la opción de crear un paquete directamente. La ventaja de hacerlo así es que cuando queramos añadir una clase directamente a ese paquete (botón derecho sobre el paquete creado, *New*, *Class...*) añadirá la clase a ese paquete directamente.

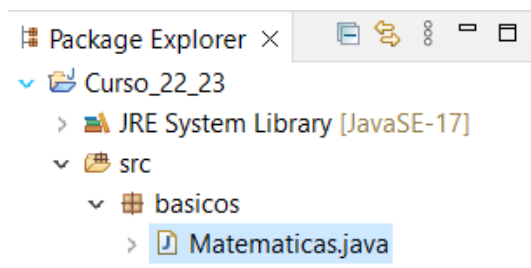
### 6.3. Ejemplo de creación y uso de paquetes propios

Para cerrar el apartado de paquetes en Java, se mostrará un ejemplo de creación y uso de paquetes. Para ello vamos a simular el paquete que contiene los paquetes básicos de java (`java.lang`) y la clase `Math` que contiene métodos para hacer cálculos matemáticos. Por supuesto, el ejemplo es una versión super reducida de la clase `Math` y con un método simple para sumar dos números enteros.

Desde el IDE de Eclipse, los pasos son:

1. Creamos un paquete llamado *basicos*.
2. Añadimos este paquete una clase llamada *Matematicas*.
3. Añadimos a esta clase un método llamado *suma*, que recibe dos parámetros de tipo entero y devuelve un valor entero que será el resultado de la suma de los dos parámetros.

Una vez realizados estos pasos, la estructura del proyecto y el código fuente deberían quedar así:



```
1 package basicos;
2
3 public class Matematicas {
4     public static int suma(int a, int b) {
5         return a + b;
6     }
7 }
```

Como puedes observar no hay un método llamado *main* en esta clase. Esto es correcto, ya que no queremos tener un punto de entrada de programa para ejecutar esta clase. La clase *Matematicas* sólo nos sirve para usar sus métodos.

Ahora si queremos probar la clase *Matematicas* podemos crear otra clase *Test* (en otro paquete) y en ella importar el paquete *basicos*. Por tanto, dicha clase quedaría así:

```
1 package ejemplos; // Situar package antes que import
2
3 import basicos.Matematicas;
4
5 public class Test {
6     public static void main(String[] args) {
7         System.out.println(Matematicas.suma(2, 3));
8     }
9 }
10 }
```

**Nota**

En el código fuente es aconsejable situar los *package* antes que los *import*.

## 7. Bibliografía

---

Documentación oficial: <https://docs.oracle.com/en/java/javase/17/docs/api/index.html>

Web w3schools.com

Apuntes de José Chamorro del CFGS DAW del IES Sant Vicent Ferrer (Algemesí).

Apuntes de: <https://github.com/statickidz/TemarioDAW/blob/master/PROG/PR05.pdf>