

# PROGRAMACIÓN

## Desarrollo de Aplicaciones Multiplataforma

*Manual del módulo*

*UD 6 – CLASES*



Ander Gonzalez Ortiz



Esta obra está bajo una Licencia Creative Commons Atribución-NoComercial-SinDerivadas 4.0 Internacional.

## 6.1. INTRODUCCIÓN

Hasta aquí hemos utilizado un paradigma de programación llamado programación estructurada, que emplea las estructuras de control (condicionales y bucles), junto a datos y funciones. Una de sus principales desventajas es que no existe un vínculo fuerte entre funciones y datos. Esto dificulta el tratamiento de problemas complejos. Por eso, llegados a este punto, vamos a saltar a un nuevo paradigma, la programación orientada a objetos (en adelante POO). que amplía los horizontes de un programador dotándolo de nuevas herramientas que facilitan la resolución de problemas complejos.

## 6.2. DEFINICIÓN DE UNA CLASE

La POO se inspira en una abstracción del mundo real, en la que los objetos se clasifican en grupos. Por ejemplo, todos los mamíferos de cuatro patas que dicen ¡guau! se engloban dentro del grupo de los perros. Si observamos a Pepa, Paco y Miguel, vemos que los tres pertenecen al mismo grupo: los tres son personas. Pepa es una persona, Paco es una persona y Miguel también es una persona. Todos pertenecen al grupo de las personas. En el argot de la POO, a cada uno de estos grupos se le denomina **clase**.

Podemos definir cada grupo o clase mediante las propiedades y comportamientos que presentan todos sus miembros. Una propiedad es un dato que conocemos de cada miembro del grupo, mientras que un comportamiento es algo que puede hacer.

Vamos a definir la clase persona mediante dos elementos:

- **Propiedades:** un nombre, una edad y una estatura. Tanto Pepa como Paco como Miguel tienen un nombre, una edad y una estatura; son datos que en cada uno tendrá un valor distinto.
- **Comportamientos:** Pepa, Paco y Miguel (en realidad todas las personas) pueden saludar, crecer o cumplir años, por ejemplo.

Como vemos, es posible definir una clase mediante un conjunto de propiedades y comportamientos. La sintaxis para definir una clase usa la palabra reservada `class`.

```
public class NornbreClase{  
    //definición de la clase  
}
```

Por ejemplo, la clase Persona se define:

```
public class Persona {  
    String nombre;  
    byte edad;  
    float estatura;  
}
```

## 6.3. CREAR UNA CLASE EN ECLIPSE

La definición de una clase tiene que escribirse en un fichero independiente que tiene que llamarse igual que la clase y tendrá extensión .java. Gracias a Eclipse no tendremos que

estar pendientes de estos detalles. La forma de crear una clase desde Eclipse es la siguiente:

1. Hacer click derecho sobre la carpeta `src` del proyecto, New > Class

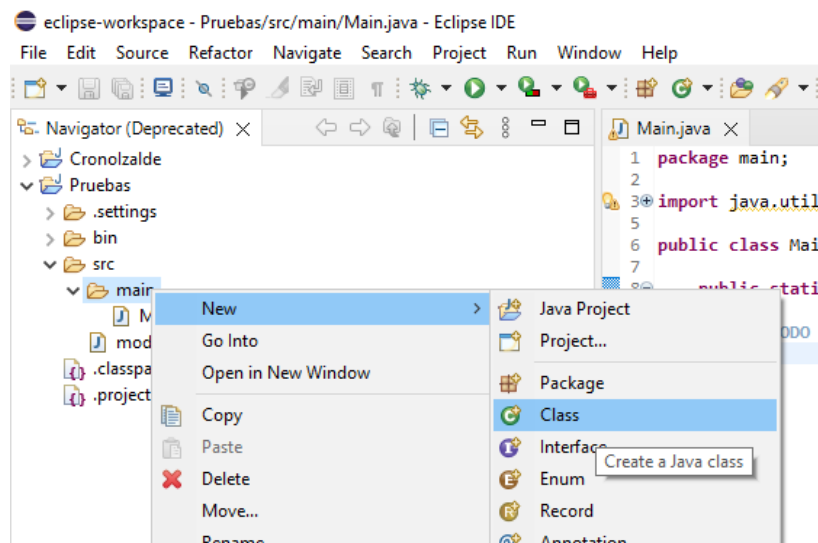


Figura 1. Crear un elemento: elegir un tipo.

2. En el campo "Name", escribimos el nombre de la clase a crear. El resto de configuraciones las dejamos por defecto.

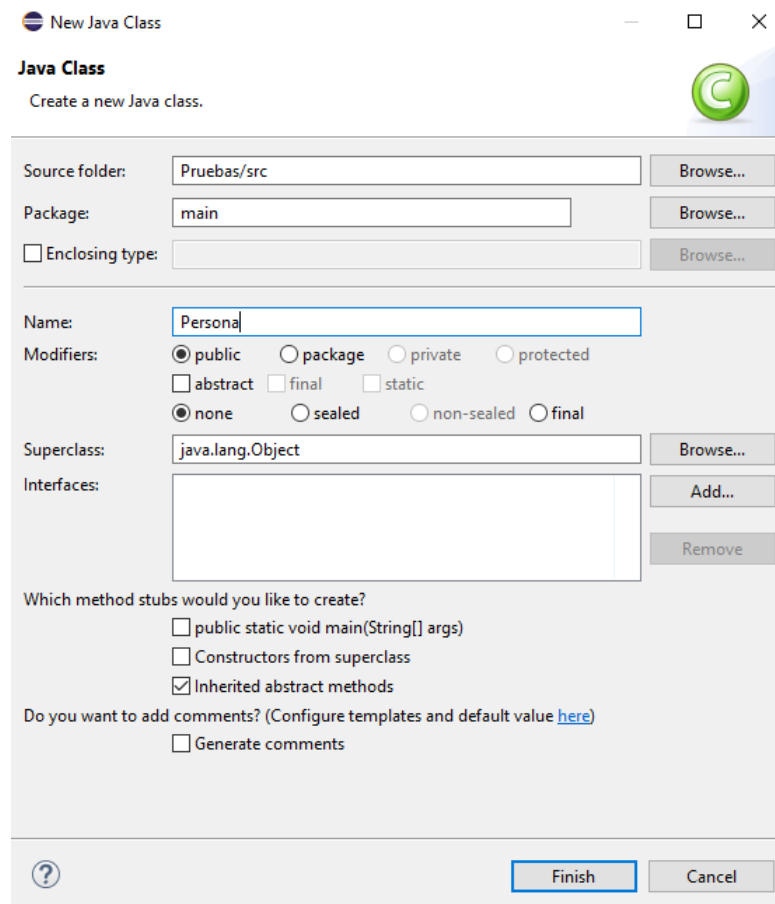


Figura 2. Características de una clase: desde su nombre, el proyecto en el que se incluye, etcétera.

3. Finalmente, con el botón "Finish" terminamos el proceso.

A partir de ahora disponemos de la nueva clase, en la que falta definir sus atributos y métodos.

#### 6.4. ATRIBUTOS

Los datos que definen una clase se denominan atributos. Por ejemplo, la clase `Vehículo` puede definirse mediante los atributos matrícula, color, marca y modelo. Como se ha visto, nuestra clase `Persona` dispone de los atributos nombre, edad y estatura.

La forma de declarar los atributos en una clase es:

```
public class NombreClase {  
    tipo atributo1;  
    tipo atributo2;  
}
```

El tipo especificado en `tipo` puede ser cualquier tipo primitivo -o una clase, como veremos a lo largo de esta unidad-. El código para definir nuestra clase `Persona` será:

```
public class Persona {  
    String nombre;  
    byte edad;  
    double estatura;  
}
```

En una clase es posible declarar atributos de varios tipos: primitivos (por ejemplo: `int edad`), de otras clases (como por ejemplo: `String nombre`) y de tipos de una interfaz. Las interfaces se estudiarán en profundidad en próximas unidades.

##### 6.4.1. Inicialización

Se puede asignar un valor por defecto a los atributos de una clase; esto se realiza en la propia declaración de la forma.

```
public class NombreClase {  
    tipo atributo1 = valor;  
    ...  
}
```

En general, los atributos pueden cambiar durante la ejecución de un programa, salvo que sean declarados con el modificador `final`. En este caso, el atributo será una constante que, una vez inicializado, no podrá cambiar de valor.

Si suponemos que el DNI de una persona no cambia una vez asignado,

```
public class Persona {
    String nombre;
    byte edad;
    double estatura;
    final String dni; //una vez asignado no podrá cambiarse
}
```

La inicialización de un atributo **final** puede hacerse en su declaración o, como veremos más adelante, por medio de un constructor.

## 6.5. OBJETOS

Los elementos que pertenecen a una clase se denominan instancias u objetos. Cada uno tiene sus propios valores de los atributos definidos en la clase. Explicaremos este concepto con un símil: supongamos que una clase es un formulario donde se solicitan una serie de datos. Cada formulario relleno recoge distintos valores para los datos que se solicitan, siendo cada uno de los formularios rellenos, en nuestro símil, un objeto concreto. Todos los formularios cumplimentados -los objetos- tendrán la misma estructura. Esto es lógico, ya que utilizamos como plantilla el mismo formulario -la clase-. Sin embargo, cada formulario relleno -objetos- tendrá distintos valores: distintos nombres, direcciones, etcétera.

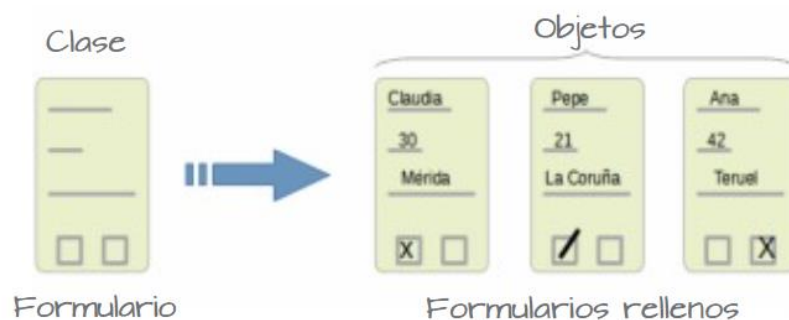


Figura 3. Formulario general y formularios rellenos.

Si nos fijamos de nuevo en Pepa, Paco y Miguel, nos damos cuenta de que cada uno de ellos es un objeto de la clase **Persona**. La Figura 7.4 muestra la clase junto a tres objetos con distintos valores para sus atributos.

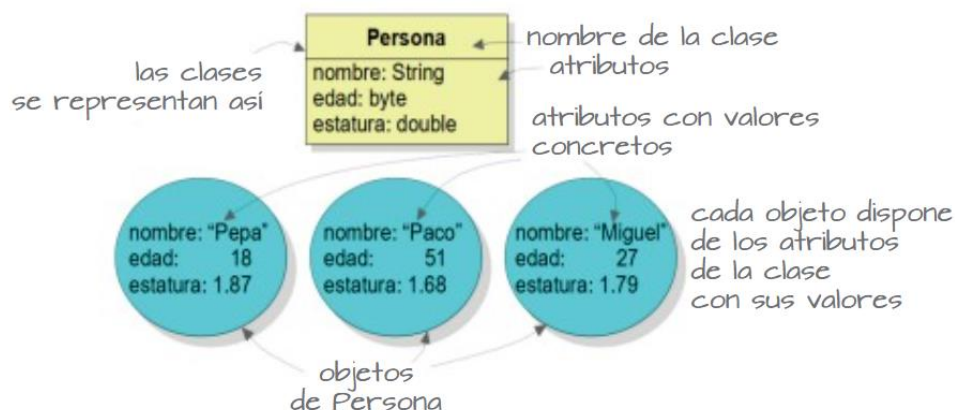


Figura 4. Ejemplo de valores de atributos para personas.

### 6.5.1. Referencias

El comportamiento de los objetos en la memoria del ordenador y sus operaciones elementales (creación, asignación y destrucción) son idénticos al de las tablas. Esto es debido a que ambos, objetos y tablas, utilizan las referencias (véase el Apartado 5.4). De hecho, las propias tablas se consideran en Java como un tipo más de objetos.

Recordemos brevemente el concepto de referencia: la memoria de un ordenador está formada por pequeños bloques consecutivos identificados por un número único que se denomina dirección de memoria. Es habitual utilizar números hexadecimales; por este motivo, las direcciones de memoria tienen un aspecto similar a: 2a139f55.

Cualquier dato almacenado en la memoria ocupará, dependiendo de su tamaño, una serie de bloques consecutivos, y puede ser identificado mediante la dirección del primero de ellos. A esta primera dirección de memoria que identifica un objeto se le denomina en Java referencia.

### 6.5.2. Variables referencia

Antes de construir un objeto necesitamos declarar una variable cuyo tipo sea su clase. La declaración sigue las mismas reglas que las variables de tipo primitivo,

```
Clase nombreVariable;
```

donde `Clase` será el nombre de cualquier clase disponible.

Veamos cómo declarar la variable `p` de tipo `Persona`:

```
Persona p; //p es una variable de tipo Persona
```

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

### 6.5.3. Operador new

La forma de crear objetos, como en las tablas, es mediante el operador `new`.

```
p = new Persona();
```

En este caso, crea un objeto de tipo `Persona` y asigna su referencia a la variable `p`.

El operador `new` primero busca en memoria un hueco disponible donde construir el objeto. Este, dependiendo de su tamaño, ocupará cierto número consecutivo de bloques de memoria. Por último, devuelve la referencia del objeto recién creado, que se asigna a la variable `p`.

Podemos comprobar qué aspecto tiene una referencia ejecutando

```
p = new Persona();  
System.out.println(p); //muestra en consola la referencia del objeto
```

A la hora de trabajar con referencias, es bastante más sencillo pensar en ellas como flechas que se dirigen desde la variable hacia el objeto (véase Figura 5).

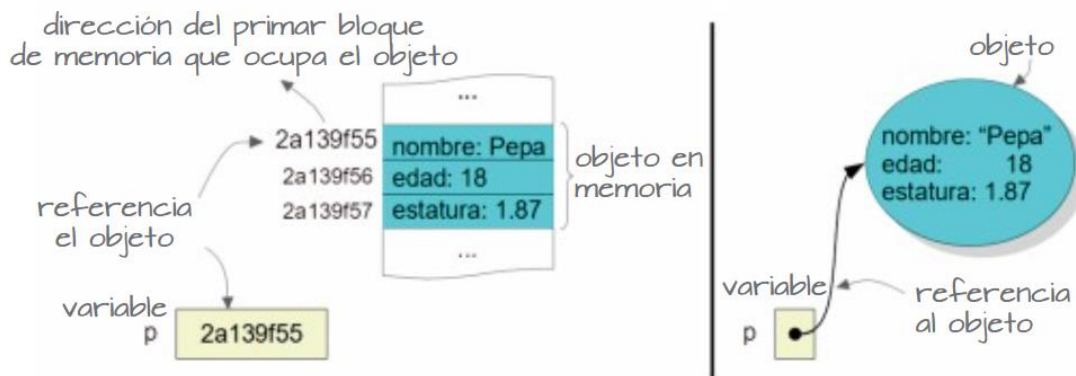


Figura 5. Dos formas de representar una misma referencia. En la izquierda, tal y como se construye el objeto en memoria. En la derecha, la representación es más intuitiva, sustituyendo la referencia por una flecha que indica a qué objeto se accede desde la variable.

En el momento en que disponemos de un objeto, podemos acceder a sus atributos mediante el nombre de la variable seguido de un punto (.) Por ejemplo, para asignar valores a los atributos del objeto referenciado por `p` escribimos:

```
p = new Persona();  
p.nombre = "Pepa";  
p.edad = 18;  
p.estatura = 1.87;
```

Es importante comprender que podemos acceder al mismo objeto mediante distintas variables que almacenen la misma referencia. La Figura 6 representa el siguiente código, donde un objeto está referenciado por dos variables:

```
Persona p1, p2;  
p1 = new Persona(); //p1 referencia al objeto creado  
p2 = p1; //asignamos a p2 la referencia contenida en p1  
p2.nombre = "Pepa" //es equivalente a utilizar p1.nombre
```

Ahora podemos acceder al objeto de dos maneras: mediante `p1` o mediante `p2`. En ambos casos estamos referenciando el mismo objeto.

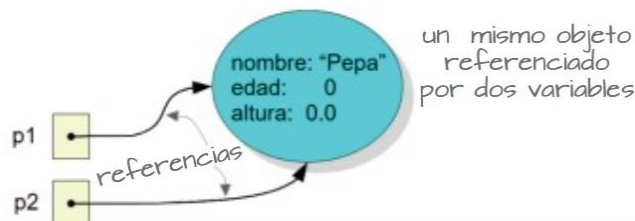


Figura 6. Un mismo objeto referenciado por dos variables.

El mecanismo en el que varias variables comparten la misma referencia es aprovechado por la clase `String` para ahorrar espacio en textos usados frecuentemente, ya que Java se

encarga por su cuenta de que todas las variables a las que se les han asignado idéntico literal cadena compartan su referencia.

```
String a = "Hola mundo";  
String b = "Hola mundo"; //las variables a y b guardan la misma referencia  
String c = "Escriba un número:";  
String d = "Escriba un número:"; //c y d comparten la misma referencia
```

El hecho de que se compartan las referencias de un mismo literal cadena es la causa de que la clase `String` sea inmutable. Si en el código anterior se permitiera modificar la cadena referenciada por `a`, se estaría modificando también el contenido de la variable `b` y de todas aquellas que estuvieran referenciando el mismo literal.

#### 6.5.4. Referencia null

El valor literal `null` es una referencia nula. Dicho de otra forma, una referencia a ningún bloque de memoria. Cuando declaramos una variable referencia se inicializa por defecto a `null`. Hay que tener mucho cuidado de no intentar acceder a los miembros de una referencia nula, ya que se produce un error que termina la ejecución del programa de forma inesperada.

```
Persona p; //se inicializa por defecto a null  
p.nombre //¡error!
```

La última instrucción genera un error del tipo: `Null pointer exception`, que significa que estamos intentando acceder a los atributos de un objeto que no existe.

El literal `null` se puede asignar a cualquier variable referencia.

```
Persona p = new Persona(); //p referencia un objeto  
...  
p = null; //p no referencia nada
```

#### 6.5.5. Recolector de basura

Existen tres formas de conseguir que un objeto no esté referenciado:

- Es posible, aunque no tenga mucho sentido, crear un objeto y no asignarlo a ninguna variable.

```
new Persona();
```

- Otra posibilidad es asignar `null` a todas las variables que contenían una referencia a un objeto.
- También podemos asignar un objeto distinto a la variable.

```
Persona p = new Persona(); //objeto 1  
p = new Persona(); //objeto 2. Ahora el objeto 1 queda sin referencia
```

En todos los casos, el objeto se queda perdido en memoria, es decir, no existe forma de acceder a él. Sin embargo, está ocupando memoria (véase Figura 7). Si este comportamiento se repite demasiado, fortuita o malintencionadamente, es posible que se



agote toda la memoria libre disponible, lo que impediría el normal funcionamiento del ordenador.



Figura 7. Objeto sin referencia.

Para evitar este problema, Java dispone de un mecanismo llamado **recolector de basura** (*garbage collector*), que se ejecuta de vez en cuando de forma transparente al usuario, y se encarga de comprobar, uno a uno, todos los objetos de la memoria. Si alguno de ellos no estuviera referenciado por ninguna variable, se destruye, liberando la memoria que ocupa.

## 6.6. MÉTODOS

Hemos declarado clases con atributos, pero también disponen de comportamientos. En el argot de la POO, a los comportamientos u operaciones que pueden realizar los objetos de una clase se les denomina métodos. Por ejemplo, las personas son capaces de realizar operaciones como saludar, cumplir años, crecer, etcétera.

Los métodos no son más que funciones que se implementan dentro de una clase. Su sintaxis es:

```
public class NombreClase {  
    ... //declaración de atributos  
  
    tipo nombreMétodo (parámetros) {  
        cuerpo del método  
    }  
}
```

La definición de un método es la de una función, sustituyendo **cuerpo del método** por un bloque de instrucciones. Hasta ahora todas las funciones -métodos de la clase **Main**- que hemos implementado han sido estáticas por razones que explicaremos más adelante. Sin embargo, en general, todos los métodos de una clase no tienen por qué ser estáticos.

Ampliamos la clase **Persona** con algunos métodos:

```

public class Persona {
    String nombre;
    byte edad;
    double estatura;

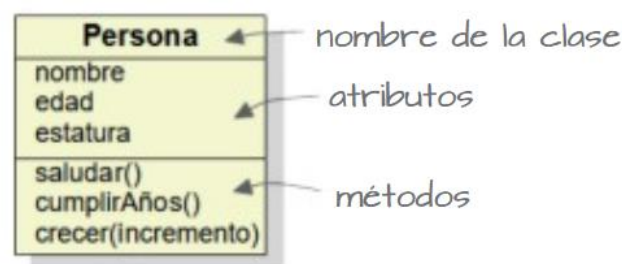
    public void saludar() {
        System.out.println("Hola. Mi nombre es " + nombre);
        System.out.println("Encantado de conocerte.");
    }

    public void cumplirAños() {
        edad++; //incrementamos la edad en 1
    }

    public void crecer(double incremento) {
        estatura += incremento; //la estatura aumenta cierto incremento
    }
}

```

La Figura 8 muestra el diagrama de clases de Persona con sus atributos y métodos.



*Figura 8. Diagrama de clases para la clase Persona. En un diagrama de clases cada clase se representa por un rectángulo subdividido en varias partes.*

A partir de ahora, los objetos de tipo Persona pueden invocar sus métodos utilizando un punto (.), al igual que se hace con los atributos. Veamos un ejemplo:

```

Persona p;
p = new Persona();
p.edad = 18;
p.cumplirAños(); //¡Felicidadse! La edad de p se incrementa
System.out.println(p.edad); //mostrará 19

```

Tanto a los atributos como a los métodos de una clase se les llama de forma genérica miembros. De esta forma, al hablar de miembros de una clase, hacemos referencia a los atributos y métodos declarados en su definición.

Los métodos de una clase tienen acceso a las siguientes variables: variables locales declaradas dentro del método, parámetros de entrada y atributos de la clase. Así mismo, tiene acceso a los demás métodos de la clase.

#### 6.6.1. Ámbito de las variables y atributos

El ámbito de una variable define en qué lugar puede usarse y coincide con el bloque en el que se declara la variable que, como se vio en el Apartado 4.2, puede ser:

- El bloque de una estructura de control: `if`, `if-else`, `switch`, `while`, `do-while` o `for`; también podemos definir bloques de usuario. Basta con poner la pareja de llaves y escribir código entre ellas. Las variables declaradas en este ámbito se denominan variables de bloque.
- Una función o método. Las variables declaradas aquí se conocen como variables locales.

Con la POO, aparece un nuevo ámbito:

- La clase. Cualquier miembro (atributo o método) definido en una clase podrá ser utilizado en cualquier lugar de ella. Los atributos son variables de la clase.

Un ámbito puede contener a otros ámbitos, formando una estructura jerárquica. Por ejemplo, una clase puede contener dos métodos, y estos, distintos bloques de, por ejemplo, una estructura `while` o `if`.

Una variable puede utilizarse en el ámbito o bloque en el que se declara, que incluye sus bloques internos. Sin embargo, no ocurre lo contrario: una variable no podrá utilizarse en el ámbito padre del bloque en el que se declara. Por ejemplo, un atributo puede emplearse dentro de un método, y una variable local dentro del bloque de una estructura de control de un método. Pero, en cambio, no podremos usar una variable local fuera de su método, ni una variable de bloque fuera de él.

El código de la Figura 9 muestra el ámbito de tres variables, donde `atributo` puede utilizarse en cualquier lugar de la clase; `varLocal` en cualquier lugar dentro del método en el que se declara; por último, `varBloque` puede usarse solo dentro del bloque de instrucciones de la estructura `while`.

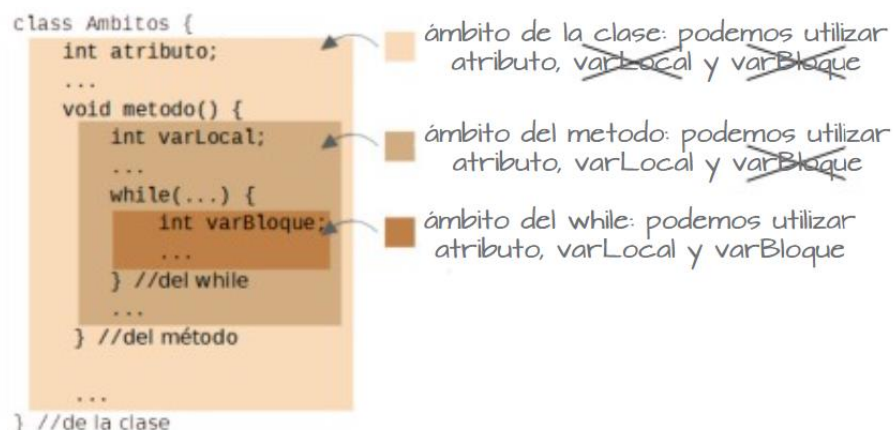


Figura 9. Representación de los distintos ámbitos de las variables y su alcance mediante un código de colores.

### 6.6.2. Ocultación de atributos

Dos variables declaradas en ámbitos anidados no pueden tener el mismo identificador, ya que esto genera un error. Sin embargo, existe una excepción cuando una variable local en un método tiene el mismo identificador que un atributo de la clase. En este caso, dentro del método, la variable local tiene prioridad sobre el atributo, es decir, que al utilizar el identificador se accede a la variable local y no al atributo. En la jerga de la POO se dice que la variable local oculta al atributo.

Veamos un ejemplo:

```
public class Ambito {
    int edad; //atributo entero
    void metodo() {
        double edad; //variable local. Oculta al atributo edad (que es entero)
        edad= 8.2; //variable local double, que oculta al atributo de la clase
        ...
    }
}
```

### 6.6.3. Objeto this

La palabra reservada `this` permite utilizar un atributo incluso cuando ha sido ocultado por una variable local. De igual manera que cada uno se refiere a sí mismo como yo, aunque tengamos un nombre que los demás utilizan para identificarnos, las clases se refieren a sí mismas como `this`, que es una referencia al objeto actual y funciona como una especie de yo para clases. Al escribir `this` en el ámbito de una clase se interpreta como la propia clase, y permite acceder a los atributos aunque se encuentren ocultos.

Estudiemos el siguiente fragmento de código donde, en el ámbito de un método, una variable local oculta un atributo de la clase:

```
public class Ambito {
    int edad; //atributo entero
    public void metodo() {
        double edad; //oculta el atributo edad (que es entero)
        edad = 20.0; //variable local, no el atributo
        this.edad = 30; //atributo de la clase
    }
}
```

## 6.7. ATRIBUTOS Y MÉTODOS ESTÁTICOS

Un atributo estático, también llamado atributo de la clase, es aquel del que no existe una copia en cada objeto. Todos los objetos de una misma clase comparten su valor.

Supongamos que necesitamos añadir a la clase `Persona` un atributo que nos indique qué día de la semana es hoy. Evidentemente si hoy es lunes, será lunes para todo el mundo; e igualmente si es martes, será martes para todos. Por tanto, el valor del atributo hoy será compartido por todos los objetos de la clase `Persona`. No tiene sentido que para una persona sea lunes y para otra sea sábado. La Figura 10 representa este concepto.

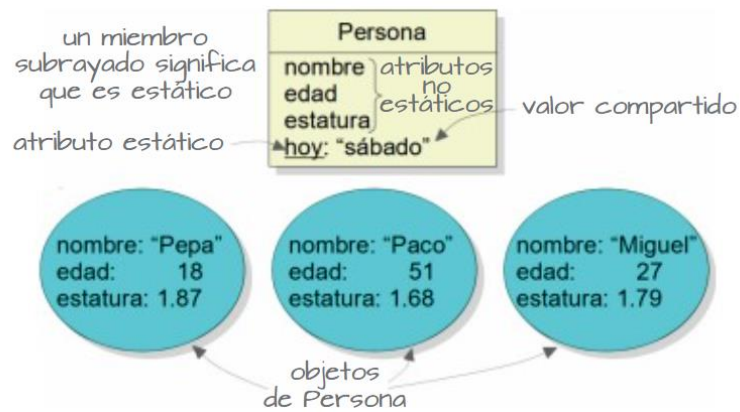


Figura 10. Diagrama de clases de Persona, con el atributo estático hoy. Dicho atributo no pertenece a los objetos, al contrario, es un atributo de la clase.

Un atributo estático se declara mediante la palabra reservada **static**.

```
public class Persona {
    ...
    static String hoy;
}
```

Para acceder a un atributo estático se utiliza el nombre de la clase de la siguiente forma:

```
Persona.hoy = "domingo";
System.out.println(Presona.hoy); //mostrará "domingo"
```

Un atributo estático se inicializa en el momento de cargar la clase en memoria; esto ocurre cuando se declara alguna variable del tipo de la clase o cuando se crea un primer objeto de dicha clase. Si deseamos asignar un valor inicial al atributo **hoy**, escribiremos

```
public class Persona {
    ...
    static String hoy = "lunes"; //valor inicial
}
```

También podemos declarar métodos estáticos. Son aquellos que no requieren de ningún objeto para ejecutarse y, por tanto, no pueden utilizar ningún atributo que no sea estático. En el caso de que lo intente, se producirá un error.

A modo de ejemplo, vamos a diseñar un método que actualice el atributo **hoy** a partir de un entero que se le pasa como parámetro. Este estará comprendido entre 1 y 7, que representan los días de la semana, de lunes a domingo.

```
public static void hoyEs(int dia) {
    hoy = switch (dia) {
        case 1 -> "lunes";
        case 2 -> "martes";
        ...
        case 7 -> "domingo";
    }
}
```

La forma de invocar un método estático es, igual que con los atributos estáticos, mediante el nombre de la clase. Vamos a actualizar el día de hoy a martes:

```
Persona.hoyEs(2); //martes
```

Por otra parte, desde un método estático solo se pueden invocar directamente métodos y atributos estáticos. Esa es la razón por la cual hasta ahora solo hemos usado métodos estáticos. Todos ellos eran invocados desde la función `main()` que siempre es `static`. No obstante, dentro de un método estático se pueden crear objetos de cualquier clase (incluida la suya propia) y desde él invocar miembros no estáticos definidos en esta clase.

## 6.8. CONSTRUCTORES

¿Qué valores toman los atributos de un objeto recién creado? Los atributos a los que no se les asigna un valor en su declaración se inicializan por defecto dependiendo de su tipo, de la siguiente manera: cero para valores numéricos primitivos y `char`, `null` para referencias y `false` para booleanos.

Sin embargo, generalmente, antes de utilizar un objeto desearemos asignar determinados valores a cada uno de sus atributos. Por ejemplo, si deseamos crear un objeto de tipo `Persona` con nombre, "Claudia", una edad de 8 años y una estatura de 1,20 m,

```
Persona p = new Persona(); //creamos el objeto
p.nombre = "Claudia"; //asignamos valores
p.edad = 8;
p.estatura = 1.20;
```

Este proceso (asignar valores) es necesario cada vez que creamos un objeto si no queremos trabajar con los valores por defecto. El operador `new` facilita esta tarea mediante los constructores. Un constructor es un método especial que debe tener el mismo nombre que la clase, se define sin tipo devuelto (ni siquiera `void`), y se ejecuta inmediatamente después de crear el objeto. El principal cometido de un constructor es asignar valores a los atributos, aunque también se puede utilizar para otros fines como crear tablas, mostrar cualquier tipo de información, crear otros objetos que necesitemos, etcétera.

Al constructor, como a cualquier otro método, se le puede pasar parámetros y se puede sobrecargar. Vamos a implementar un constructor para `Persona` que asigne los valores iniciales de sus atributos: `nombre`, `edad` y `estatura`:

```
public class Persona {
    ...
    public Persona(String nombre, int edad, double estatura) {
        this.nombre = nombre; //asigna el parámetro al atributo
        this.edad = edad;
        this.estatura = estatura;
    }
    ...
}
```

La llamada al constructor con los valores de los parámetros de entrada se hace por medio del operador `new`. Si deseamos crear un objeto `Persona` con los datos anteriores,

```
Persona p = new Persona("Claudia", 8, 1.20); //creamos el objeto
//y lo inicializamos mediante el constructor
```

Los atributos declarados como `final` también se pueden inicializar pasando sus valores como parámetros al constructor; no es necesario hacerlo en el sitio donde se declaran.

A la hora de sobrecargar un método tenemos que asegurarnos de que se pueda distinguir entre las distintas versiones mediante el número o el tipo de parámetros de entrada. La sobrecarga de constructores es útil cuando necesitamos inicializar objetos de varias formas. Hemos visto un constructor de `Persona` que permite asignar valores a todos los atributos. Podría darse el caso de que solo nos interesara pasar al constructor el nombre de la persona, dejando que el resto de los atributos se inicializaran con algunos valores arbitrarios.

```
public class Persona {
    ...
    //constructor que asigna valores a todos los atributos
    public Persona (String nombre, int edad, double estatura) {
        this.nombre = nombre;
        this.edad = edad;
        this.estatura = estatura;
    }

    //constructor sobrecargado: solo asigna el nombre
    public Persona (String nombre) {
        this.nombre = nombre;
        estatura = 1.0;    //valor arbitrario para la estatura
        //al no asignar la edad se inicializa por defecto: a 0
    }
}
```

Ahora disponemos de dos constructores, que se utilizan de la forma:

```
Persona a = new Persona("Pepe", 20, 1.90);
Persona b = new Persona("Dolores");
```

Cuando en una clase no se implementa ningún constructor, Java se encarga de crear uno que se denomina constructor por defecto. Este no usa parámetros de entrada e inicializa los atributos a cero, false o null según el tipo si no están ya inicializados en su declaración. No obstante, es conveniente implementar los constructores y no dejarlo en manos de Java. En cuanto se implementa un constructor en una clase, el constructor, por defecto, deja de estar disponible.

Un ejemplo: supongamos que definimos la clase `Mascota` sin ningún constructor; gracias al constructor, por defecto, podremos crear objetos de tipo `Mascota`:

```
Mascota perro= new Mascota();
```

### 6.8.1. this()

Cuando una clase dispone de un conjunto de constructores sobrecargados, es posible que un constructor invoque a otro y así reutilice su funcionalidad. Para eso se usa el constructor genérico `this()`, en lugar del constructor por su nombre. La forma de distinguir los distintos constructores, igual que en cualquier método sobrecargado, es mediante el número y el tipo de los parámetros de entrada.

Vamos a redefinir el constructor de `Persona` al que solo se le pasa el nombre usando `this()`:

```
public class Persona {  
    ...  
    //constructor que asigna valores a todos los atributos  
    public Persona (String nombre, int edad, double estatura) {  
        this.nombre = nombre;  
        this.edad = edad;  
        this.estatura = estatura;  
    }  
  
    //constructor sobrecargado que solo asigna el nombre  
    public Persona (String nombre) {  
        this(nombre, 0, 1.0); //invoca al primer constructor  
        //la edad se pone a 0 y la estatura a 1.0  
    }  
}
```

Tenemos que tener presente que, en el caso de utilizar `this()`, tiene que ser siempre la primera instrucción de un constructor; en otro caso se producirá un error.

## 6.9. PAQUETES

En Java es importante controlar la accesibilidad de unas clases desde otras por razones de seguridad y eficiencia. Esto se consigue mediante paquetes, que son contenedores que permiten guardar clases en compartimentos separados, de modo que podamos decidir, por medio de la importación, qué clases son accesibles y cómo se accede a ellas desde una clase que estemos implementando.

Todas las clases están dentro de algún paquete que, a su vez, pueden estar anidados, unos dentro de otros. Se considera que una clase que pertenece a un paquete, que a su vez está dentro de otro, solo pertenece al primero, pero no al segundo.

Un archivo fuente de Java es un archivo de texto con extensión `.java`, que se guarda en un paquete y que contiene los siguientes elementos:

- Una sentencia donde se especifica el paquete al que pertenece, que empieza con la palabra clave `package` seguida del nombre del paquete.
- Una serie opcional de sentencias de importación, con la palabra reservada `import`, que permite importar clases definidas en otros paquetes.



- La definición de una o más clases, de las cuales solo una puede ser declarada pública (por medio del modificador de acceso `public`). De todas formas, es recomendable que en cada archivo fuente se defina una sola clase, que debe tener el mismo nombre que el archivo.

### 6.9.1. Crear un paquete desde Eclipse

En Java, cada paquete se convierte físicamente en un directorio que contiene clases y otros paquetes. Gracias a Eclipse no tendremos que estar pendientes de la ubicación ni de la estructura de estos directorios. La forma de crear un paquete desde Eclipse es la siguiente:

1. Haz click sobre la carpeta donde se desea crear el paquete, New > Package, que da acceso a la ventana que se muestra en la Figura 7.11



Figura 11. Crear un elemento: elegir el tipo.

2. En la venta emergente escribir el nombre del paquete.

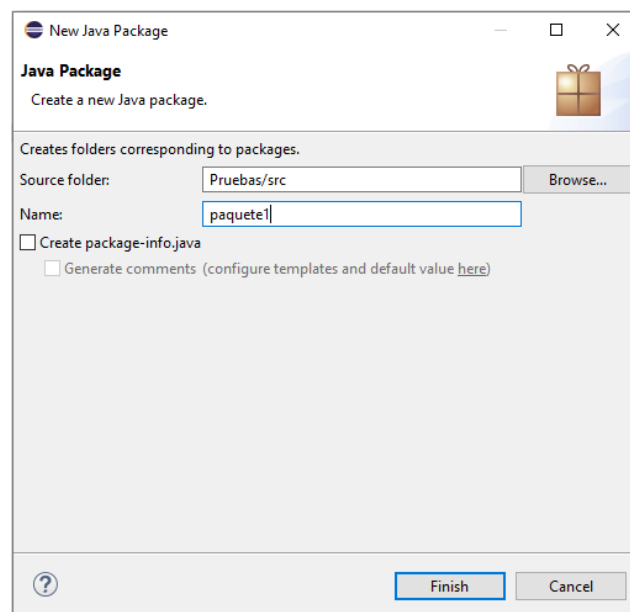


Figura 12. Propiedades del paquete.

3. Por último, mediante el botón Finish, terminamos el proceso.

A partir de ahora disponemos del nuevo paquete, en el que falta incluir las clases que contendrá. La estructura de paquetes de Eclipse queda como se muestra en la Figura 13.

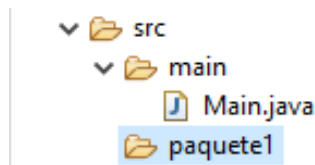


Figura 13.

*Figura 7.13. Estructura de paquetes resultante tal y como aparece en la ventana de proyectos.*

Otra opción para crear un paquete es utilizar el ratón. Para ello seleccionamos sobre *Source Packages* o cualquier paquete de la lista y pulsamos el botón derecho del ratón. Esto permite acceder a un menú contextual con el que podemos crear, entre otras cosas, nuevos paquetes.

## 6.10. MODIFICADORES DE ACCESO

Una clase será visible por otra, o no, dependiendo de si se ubican en el mismo paquete y de los modificadores de acceso que utilice. Estos modifican su visibilidad, permitiendo que se muestre u oculte.

De igual manera que podemos modificar la visibilidad entre clases, es posible modificar la visibilidad entre los miembros de distintas clases, es decir, qué atributos y métodos son visibles para otras clases.

### 6.10.1. Modificadores de acceso para clases

Debido a la estructura de clases, organizadas en paquetes, que utiliza Java, dos clases cualesquiera pueden definirse de las siguientes formas (véase Figura 14):

- Clases vecinas: cuando ambas pertenecen al mismo paquete.
- Clases externas: cuando se han definido en paquetes distintos.



Figura 14.

*Figura 14. Representación de clases vecinas y externas. A la izquierda, representación en un diagrama de clases; en la derecha con estructura de árbol, como aparece en el navegador de proyectos de Eclipse.*

Una aplicación puede entenderse como un conjunto de instrucciones que usan los servicios proporcionados por otras clases para resolver un problema. Para conocer qué servicios o herramientas están disponibles, las clases siguen el lema “si lo ves, puedes utilizarlo”.

### Visibilidad por defecto

Cuando definimos una clase sin utilizar ningún modificador de acceso,

```
package miprimerprograma.paquetel;  
class B { // sin modificador de acceso  
    ...  
}
```

se dice que usa visibilidad por defecto, que hace que solo sea visible por sus clases vecinas. En nuestro caso, B es visible por C, pero no será visible por A (Figura 15) .

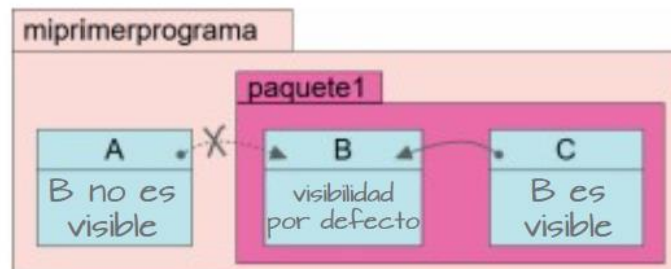


Figura 15. Visibilidad de la clase B desde A y C.

### Visibilidad total

En la Figura 15 la clase B es invisible para A y para todas las clases externas. ¿Cómo podemos hacer que B sea visible desde A? Mediante el modificador de acceso **public**, la clase B, además de ser visible para sus vecinas, lo será desde cualquier clase externa usando una sentencia de importación. De esta forma, el modificador **public** proporciona visibilidad total a la clase.

Vamos a redefinir B para que tenga visibilidad total:

```
package miprimerprograma.paquetel;  
public class B { // clase marcada como pública  
    ...  
}
```

A partir de ahora, cualquier clase, vecina o externa, puede crear objetos o acceder a los miembros públicos de B. Lo único que necesita una clase externa, como A, para acceder a B es importarla.

```
package miprimerprograma;  
import miprimerprograma.paquetel.B; //ahora A puede usar la clase B  
class A { // clase marcada como pública  
    ...  
}
```

Se importan las clases, no los paquetes. Si queremos importar todas las clases públicas de un paquete en una sola sentencia de importación, se usa el asterisco.

```
package miprimerprograma;
import miprimerprograma.paquete1.*; //A puede usar cualquier clase pública
                                     //del paquete

class A {
    ...
}
```

La visibilidad entre clases puede resumirse como: una clase siempre será visible por sus clases vecinas. Que sea visible (previa importación) por clases externas dependerá de si está declarada como pública.

	Visible desde...	
	clases vecinas	clases externas
sin modificador	✓	
public	✓	✓

#### 6.10.2. Modificadores de acceso para miembros

De igual manera que es posible modificar la visibilidad de una clase, podemos regular la visibilidad de sus miembros. Que un atributo sea visible significa que podemos acceder a él, tanto para leer como para modificarlo. Que un método sea visible significa que puede ser invocado.

Para que un miembro sea visible, es indispensable que su clase también lo sea. Es evidente que, si no podemos acceder a una clase, no existe forma alguna de acceder a sus miembros.

Debemos destacar que cualquier miembro es siempre visible dentro de su propia clase, indistintamente del modificador de acceso que utilicemos. Es decir, desde dentro de la definición de una clase siempre tendremos acceso a todos sus atributos y podremos invocar cualquiera de sus métodos.

```
public class A { //clase pública
    int dato; //su ámbito es toda la clase:
    ... // el atributo dato es accesible desde cualquier lugar de A
}
```

#### Visibilidad por defecto

Cuando queramos acceder a miembros de otra clase hay diversos grados de visibilidad. La visibilidad por defecto es aquella que se aplica a miembros declarados sin ningún modificador de acceso, como el atributo dato en el código anterior.

La visibilidad por defecto hace que un miembro sea visible desde las clases vecinas, pero invisible desde clases externas.

En nuestro ejemplo, el atributo `dato` será:

- Visible por clases vecinas, que pueden acceder tanto a la clase `A` como al atributo `dato`. Acceder a una clase significa utilizar sus miembros visibles, incluidos los constructores que permiten crear objetos.
- Invisible desde clases externas. Cualquier clase externa podrá acceder a la clase `A` - previa importación- por ser pública, pero no al atributo `dato`.

No olvidemos que la clase `A` se ha definido `public`, lo que permite que sea visible desde clases externas. Si `A` no fuera visible desde el exterior, tampoco lo serían sus miembros, sin importar el modificador utilizado en su declaración.

### Modificador de acceso `private` y `public`

Con el modificador `private` obtenemos una visibilidad más restrictiva que por defecto, ya que impide el acceso incluso para las clases vecinas. Un miembro, ya sea un atributo o un método, declarado privado es invisible desde fuera de la clase.

En cambio, `public` hace que un miembro sea visible incluso desde clases externas previa importación. Otorga visibilidad total.

El uso de `private` está justificado cuando queremos controlar los cambios de un atributo o cuando deseamos que no se conozca directamente su valor, o bien cuando queremos que un método solo sea invocado desde otros métodos de la clase, pero no fuera de ella. El acceso a esos miembros privados deberá hacerse a través de algún método `public` de la misma clase.

En el siguiente ejemplo se implementa la clase `Alumno` con los atributos `nombre` y `nota media`. Esta última será un atributo privado, ya que interesa controlar el rango de valores válidos, que estarán comprendidos entre 0 y 10, inclusive. El método público `asignaNota()` será el encargado de controlar el valor asignado a la nota.

```
public class Alumno {
    public String nombre; //atributo público
    private double notaMedia; //atributo privado
    String direccion; //atributo con visibilidad por defecto

    public void asignaNota(double notaMedia) {
        //nos aseguramos de que esté en el rango 0...10
        if (notaMedia < 0 && notaMedia > 10) {
            System.out.println("Nota incorrecta");
        } else {
            this.notaMedia = notaMedia;
        }
    }
}
```

De este modo, solo es posible modificar la nota a través del método que la controla.

La siguiente tabla muestra un resumen del alcance de la visibilidad de los miembros de una clase, según el modificador de acceso que se utilice.

	Visible desde...		
	la propia clase	clases vecinas	clases externas
private	✓		
sin modificador	✓	✓	
public	✓	✓	✓

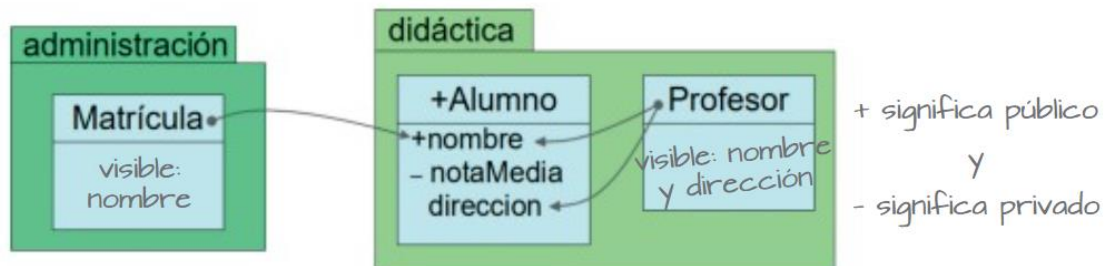


Figura 16. Ejemplo de clases con visibilidad private, public y por defecto.

### 6.10.3. Métodos get/set

Un atributo público puede ser modificado desde cualquier clase, lo que a veces tiene sus inconvenientes, ya que es imposible controlar los valores asignados, que pueden no tener sentido. Por ejemplo, nada impide que se asigne a un atributo edad un valor negativo.

Por este motivo, existe una convención en la comunidad de programadores que consiste en ocultar atributos y, en su lugar, crear dos métodos públicos: el primero (habitualmente llamado set) permite asignar un valor al atributo, controlando el rango válido de valores. Y el segundo (habitualmente llamado get) devuelve el atributo, lo que posibilita conocer su valor. Los métodos set/get hacen, en la práctica, que un atributo no visible se comporte como si lo fuera.

Estos métodos se identifican con set/get seguido del nombre del atributo. Para el atributo **edad** quedaría:

```
public class Persona {
    private int edad;
    ...

    public void setEdad(int edad) {
        if (edad >= 0) { // solo los valores positivos tienen sentido
            this.edad = edad;
        } //encaso contrario no se modifica la edad
    }

    public int getEdad() {
        return edad;
    }
}
```

Las ventajas de utilizar métodos set/get son que la implementación de la clase se encapsula, ocultando los detalles y, por otro lado, permite controlar qué atributos son accesibles para lectura y cuáles para escritura, así como los valores asignados. En nuestro ejemplo se ha limitado el uso a valores no negativos para la edad.

### 6.11. ENUMERADOS

Los tipos enumerados sirven para definir grupos de constantes como posibles valores de una variable. Por ejemplo, `DiaDeLaSemana` sería un tipo enumerado que puede tomar solo los valores constantes: LUNES, MARTES... DOMINGO. Se define de forma parecida a una clase:

```
enum DiaDeLaSemana {  
    LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO  
}
```

En la definición usamos la palabra clave `enum` y no `class`. En un programa se accede a sus valores de la forma `DiaDeLaSemana.LUNES`, `DiaDeLaSemana.MARTES`, etcétera.

Un tipo enumerado se puede implementar en un archivo aparte (normalmente dentro del mismo paquete, aunque no es obligatorio), como si fuera una clase o bien dentro de la definición de la clase donde se va a usar. En el primer caso, en Eclipse pulsamos con el botón derecho sobre el nombre del paquete: `New/JavaEnum`.

Ahora, si queremos guardar en una variable el día de la semana que tenemos inglés (los lunes) escribiremos:

```
DiaDeLaSemana ingles = DiaDeLaSemana.LUNES;
```

Normalmente, cuando tengamos que introducir por teclado un valor de tipo enumerado, escribiremos una cadena como "LUNES", y no "DiaDeLaSemana.LUNES". Para asignarlo a una variable de tipo `DiaDeLaSemana`, tendremos que convertirla en el valor enumerado correspondiente. Para eso se usa el método `valueOf()`, que convierte la cadena "LUNES" en el valor `DiaDeLaSemana.LUNES`.

```
Scanner sc = new Scanner(System.in);  
String dia = sc.nextLine(); //introducimos LUNES  
DiaDeLaSemana ingles= DiaDeLaSemana.valueOf(dia);
```

Si vamos a usar un tipo enumerado exclusivamente dentro de una clase, se puede definir dentro de ella. Por ejemplo, para añadir a la clase `Cliente` el atributo `sexo` con los valores posibles HOMBRE y MUJER, definimos el tipo enumerado `Sexo` dentro de la clase:

```
class Cliente {  
    enum Sexo {HOMBRE, MUJER}; //definición del tipo enumerado  
    Sexo sexo; //declaración de un atributo del tipo enumerado  
    ...  
}
```

Aquí, el tipo `Sexo` solo es accesible directamente desde dentro de la propia clase. Al escribir el constructor de `Cliente`, tenemos dos opciones para el parámetro de entrada del atributo `sexo`:

1. Definirlo de tipo `String` y convertirlo en `Sexo` dentro del código del constructor.

```
Cliente(..., String sexo) {  
    ...  
    this.sexo = Sexo.valueOf(sexo);  
}
```

2. Definimos el tipo `Sexo` directamente.

```
Cliente(..., Sexo sexo) {  
    ...  
    this.sexo = sexo;  
}
```

En el primer caso, cuando se llame al constructor, se le pasa una cadena.

```
String sexoCliente = new Scanner(System.in).next();  
Cliente c = new Cliente(..., sexoCliente);
```

En segundo caso, habrá que hacer la conversión de `String` a `Sexo` antes de llamar al constructor. Dicha conversión dependerá de dónde está definido el tipo enumerado.

- a) Si está definido dentro de la clase `Cliente`, se accede a él con el nombre de la clase,

```
Cliente c = new Cliente(..., Cliente.Sexo.valueOf(sexoCliente));
```

- b) Si se ha definido en un archivo propio, aunque dentro del mismo paquete,

```
Cliente c = new Cliente(..., Sexo.valueOf(sexoClienetne));
```

Los tipos enumerados se pueden definir en paquetes distintos a donde se vayan a usar. En ese caso, habrá que definirlos `public` e importarlos igual que si fueran clases.