

U.D.7: CLASES

Basado en el libro de
Paraninfo

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.

INTRODUCCIÓN

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.

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.

DEFINICIÓN DE UNA CLASE

Nota: Una persona es mucho más que un nombre, una edad y una estatura, pero estamos haciendo una abstracción, donde elegimos las propiedades que nos interesan.

DEFINICIÓN DE UNA 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***
- ***Comportamientos***

DEFINICIÓN DE UNA CLASE

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.

DEFINICIÓN DE UNA CLASE

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**.

```
class NombreClase {  
    //definición de la clase  
}
```

Por ejemplo, la clase **Persona** se define:

```
class Persona {  
    //definición de Persona  
}
```


DEFINICIÓN DE UNA CLASE

Aclaración sobre Notación: En todos los identificadores usaremos la notación **Camel**, pero para distinguir las variables de la clases, los identificadores de las primeras comenzarán en minúsculas mientras que los identificadores de las clases comenzarán con la primera letra en mayúscula.

CREAR UNA CLASE DESDE ECLIPSE

Se hará de igual forma que cuando creamos la clase principal de nuestro programa.

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 hemos visto antes, nuestra clase ***Persona***, dispone de los ***atributos nombre, edad y estatura***.

La forma de declarar los atributos en una clase es:

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

ATRIBUTOS

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:

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

ATRIBUTOS

Aclaración: 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 la Unidad 9.

ATRIBUTOS. Inicialización

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

```
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.

ATRIBUTOS. Inicialización

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

```
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.

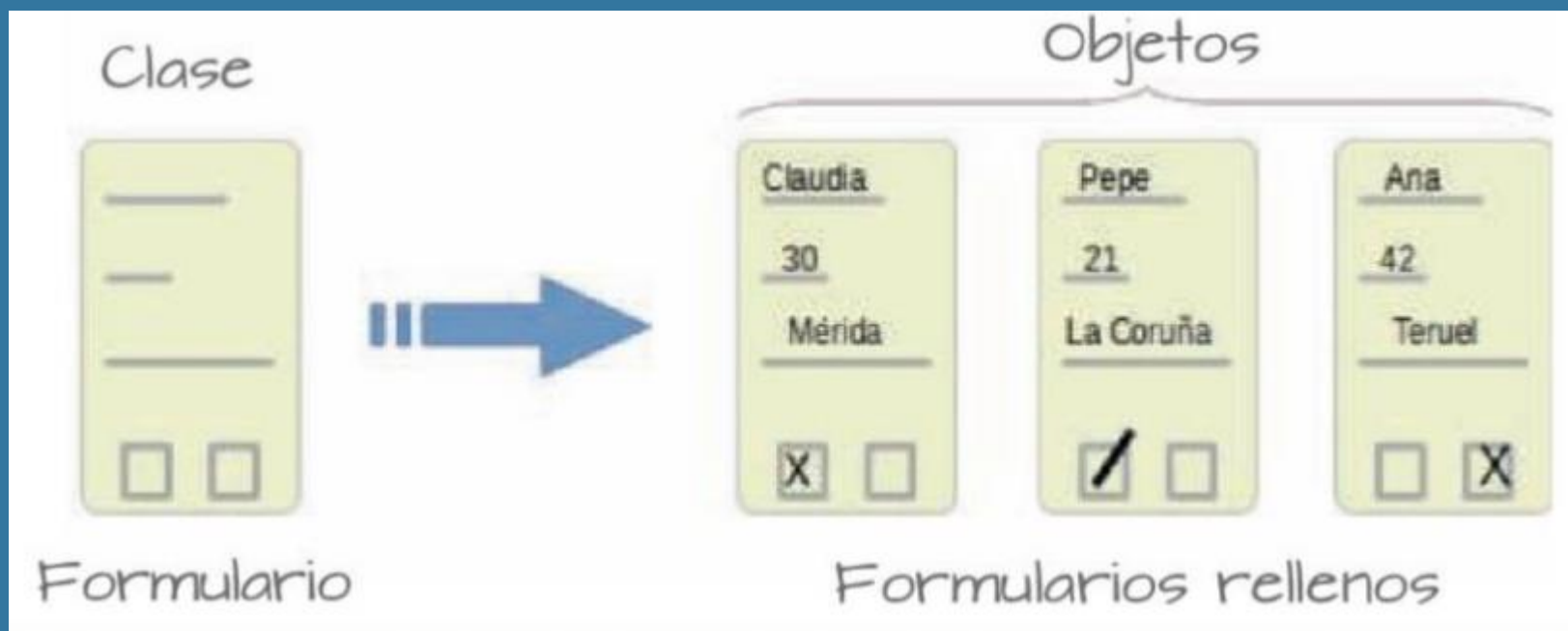
OBJETOS

Los elementos que pertenecen a una clase se denominan instancias u objetos. Cada uno tiene sus propios valores para 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.

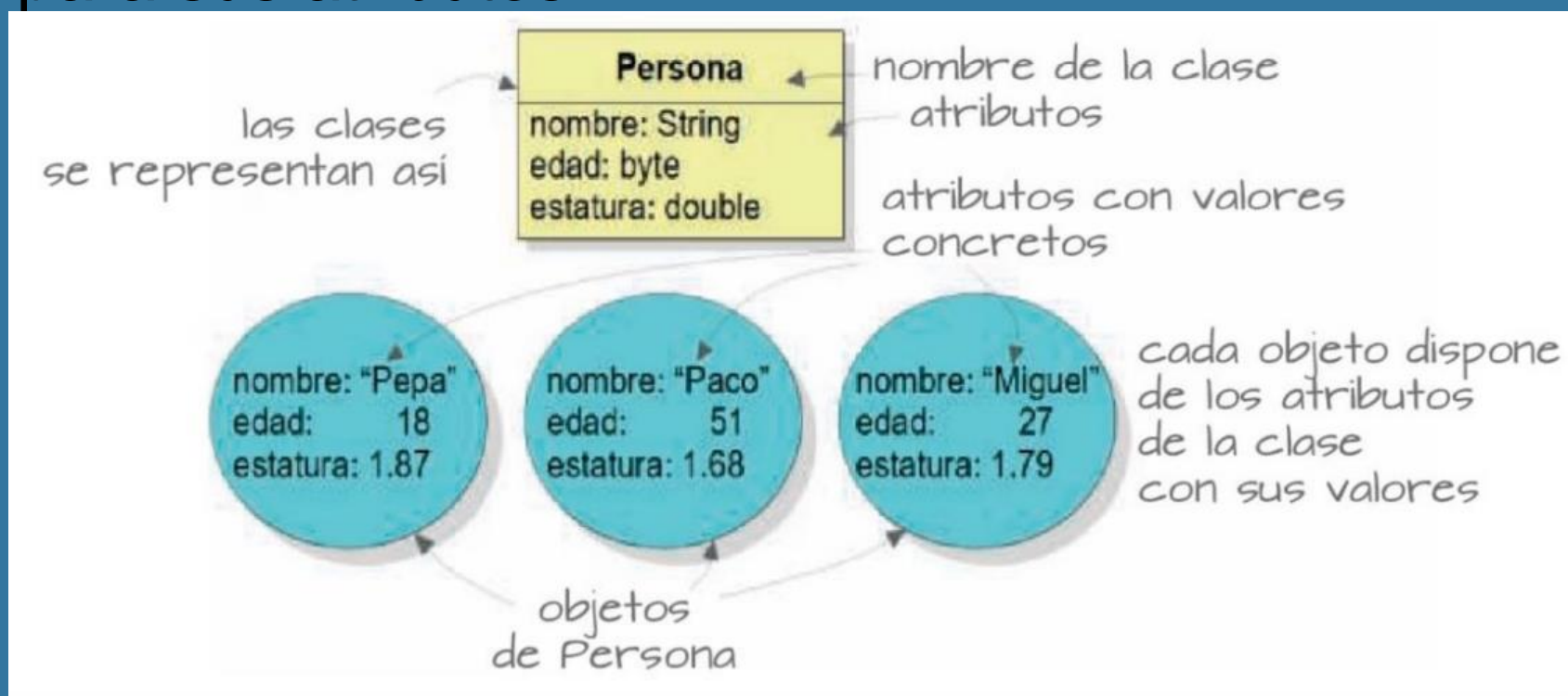
OBJETOS

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.



OBJETOS

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 siguiente figura, muestra la clase junto a tres objetos con distintos valores para sus atributos.



OBJETOS

Nota: Para representar las clases y las relaciones entre ellas se utilizan los diagramas de clases.

OBJETOS. 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. De hecho, las propias tablas se consideran en Java como un tipo más de objetos.

OBJETOS. Referencias

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 hexadecimales; por este motivo, las direcciones de memoria tienen un aspecto similar a: **2a139f55**.

OBJETOS. Referencias

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***.

OBJETOS. 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.

OBJETOS. Variables referencia

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.

OBJETOS. 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**.

OBJETOS. Operador new

Nota: El tamaño de una clase depende del tipo y la cantidad de atributos que esta contenga.

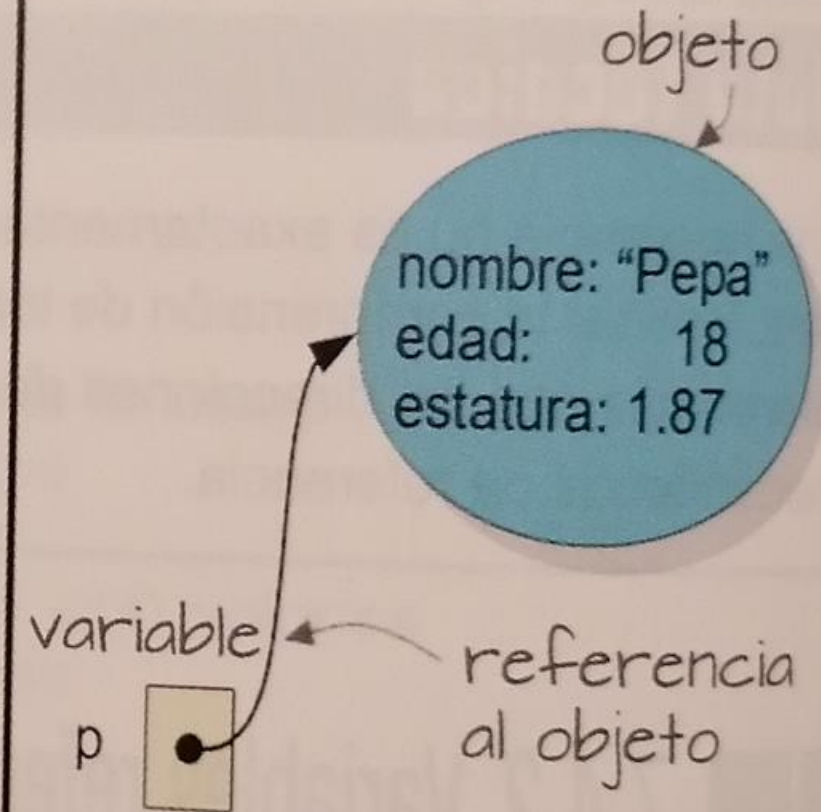
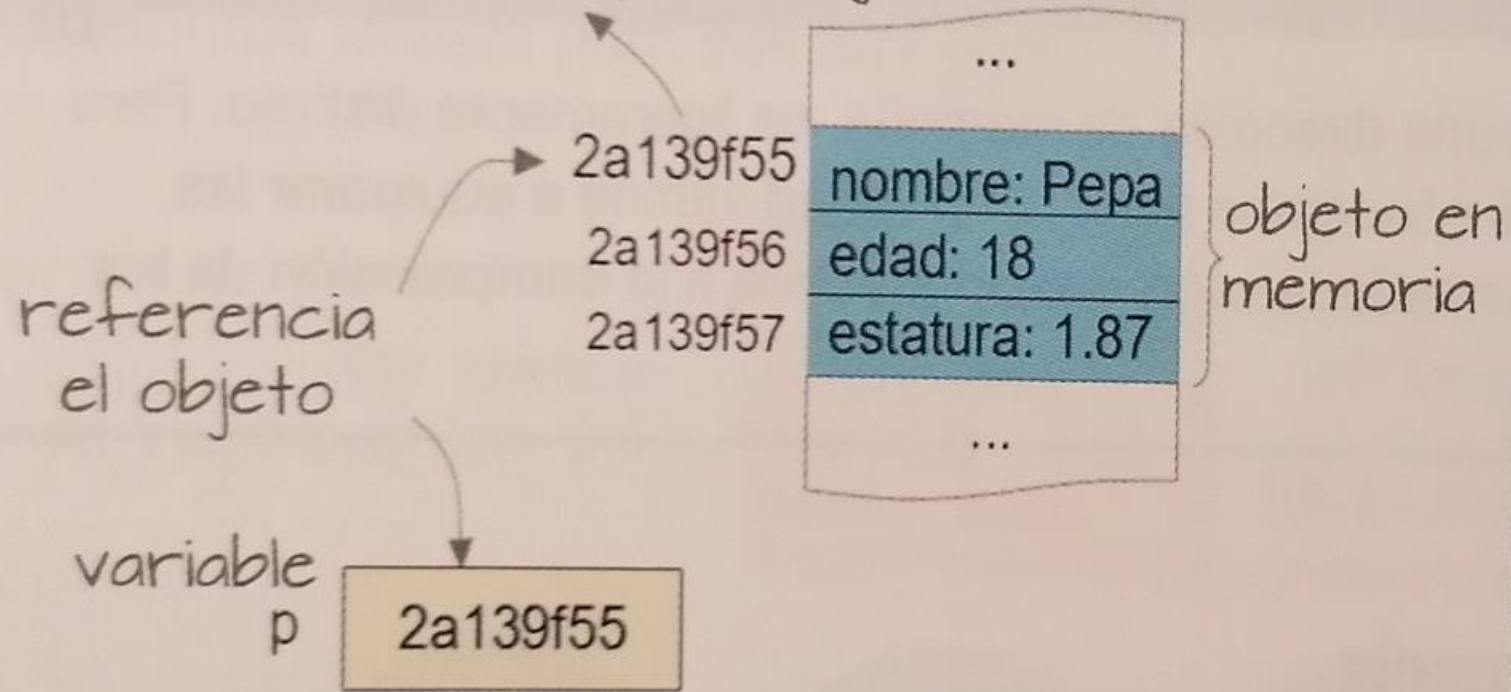
Podemos comprobar qué aspecto tiene una referencia ejecutando

```
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 (ver la siguiente figura).

OBJETOS. Operador new

dirección del primer bloque
de memoria que ocupa el objeto



OBJETOS. Operador new

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;
```

OBJETOS. Operador new

Es importante comprender que podemos acceder al mismo objeto mediante distintas variables que almacenan la misma referencia. En la siguiente figura, podemos ver el siguiente código representado, donde un objeto está referenciado por dos variables:

```
Persona p1, p2;
```

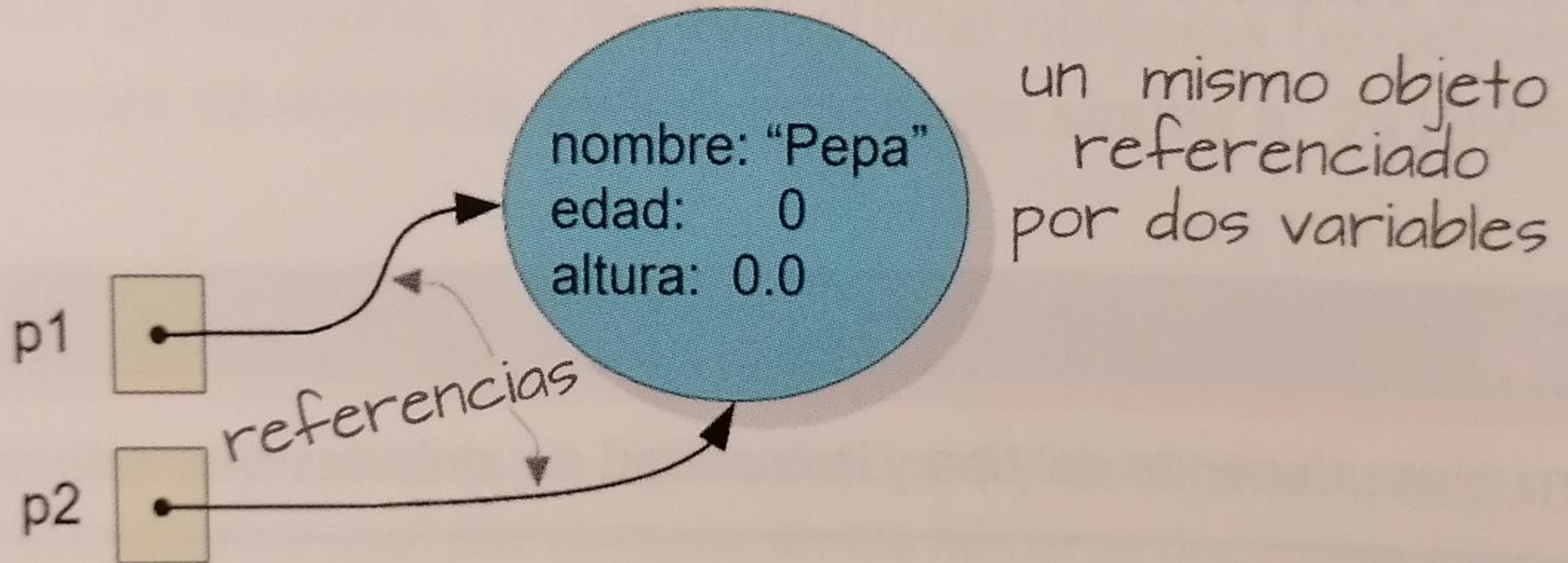
```
p1 = new Persona( ); // p1 referencia al objeto creado
```

```
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.

OBJETOS. Operador new



OBJETOS. Operador new

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

OBJETOS. Operador new

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

OBJETOS. Operador new

El hecho de que se compartan las referencias de un mismo literal cadena es la causa de que la clase 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.

OBJETOS. 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 de forma inesperada.

OBJETOS. Referencia null

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

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

OBJETOS. Referencia null

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

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

```
...
```

```
p = null // p no referencia nada
```

OBJETOS. 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.

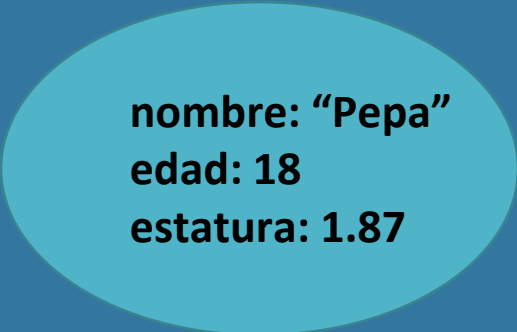
Persona p = new Persona(); // objeto 1

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

OBJETOS. Recolector de basura

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 (ver siguiente figura). 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.

OBJETOS. Recolector de basura



nombre: "Pepa"
edad: 18
estatura: 1.87

objeto sin ninguna
referencia

OBJETOS. Recolector de basura

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.

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.

MÉTODOS

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  
    }  
}
```

MÉTODOS

La definición de un método es la de una función, sustituyendo ***cuero 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.

MÉTODOS

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

```
public class Persona {  
    String nombre;  
    byte edad;  
    double estatura;  
  
    void saludar( ) {  
        System.out.println("Hola. Mi nombre es " + nombre);  
        System.out.println("Encantado de conocerte");  
    }  
}
```

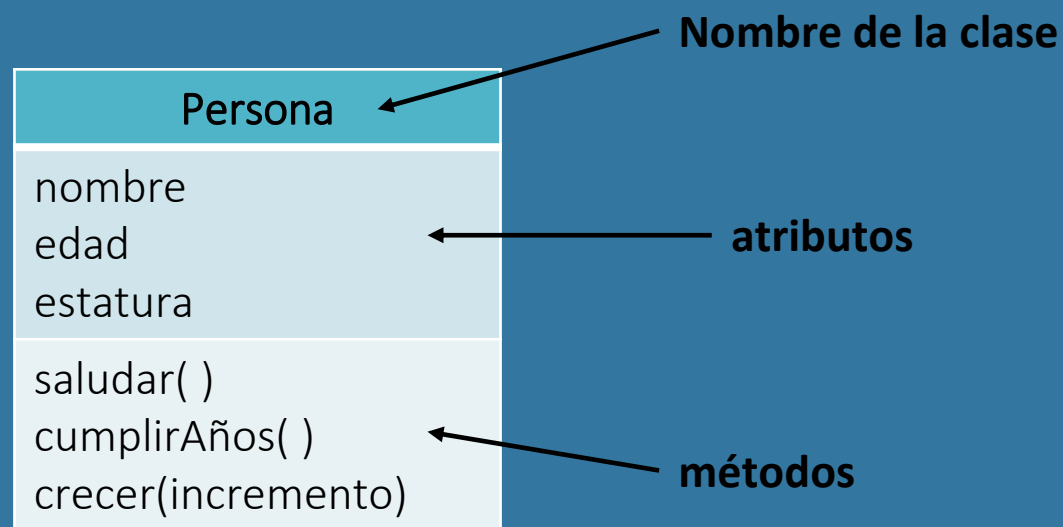
MÉTODOS

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

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

MÉTODOS

En la siguiente figura se muestra el diagrama de clases de **Persona** con sus atributos y métodos.



MÉTODOS

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( ); // ¡Felicidades! La edad de p se incrementa  
System.out.println(p.edad); // mostrará 19
```

MÉTODOS

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. Asimismo, tiene acceso a los demás métodos de la clase.

MÉTODOS. 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 vimos en su momento, 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***.

MÉTODOS. Ámbito de las variables y atributos

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**.

MÉTODOS. Ámbito de las variables y atributos

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 podemos usar una variable local fuera de su método, ni una variable de bloque fuera de él.

MÉTODOS. Ámbito de las variables y atributos

En la siguiente figura se 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**.

MÉTODOS. Ámbito de las variables y atributos

```
class Ambitos {  
    int atributo;  
    ...  
    void metodo() {  
        int varLocal;  
        ...  
        while(...) {  
            int varBloque;  
            ...  
        } //del while  
        ...  
    } //del método  
    ...  
} //de la clase
```

■ ámbito de la clase: podemos utilizar atributo, ~~varLocal~~ y ~~varBloque~~

■ ámbito del metodo: podemos utilizar atributo, varLocal y ~~varBloque~~

■ ámbito del while: podemos utilizar atributo, varLocal y varBloque

MÉTODOS. 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:

MÉTODOS. Ocultación de atributos

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

MÉTODOS. Objetos *this*

La palabra reservada **this** permite utilizar un atributo incluso cuando ha sido ocultado por la 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.

MÉTODOS. Objetos this

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

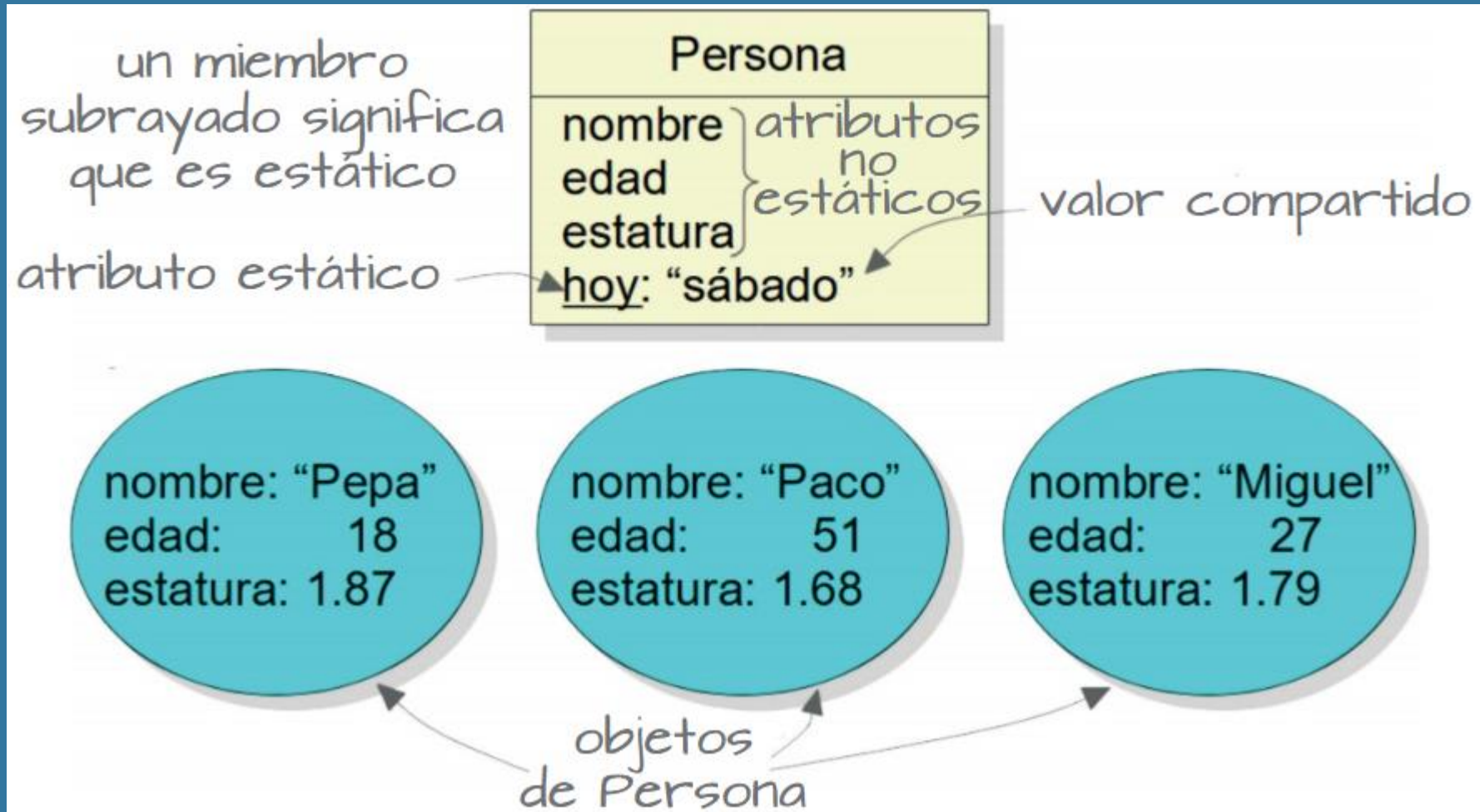
```
public class Ambito {  
    int edad; // atributo entero  
    void método( ) {  
        double edad; // oculta al atributo edad (que es entero)  
        edad = 20.0; // variable local, no el atributo  
        this.edad = 30; // atributo de la clase  
    }  
}
```

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. En la siguiente figura se representa este concepto.

ATRIBUTOS Y MÉTODOS ESTÁTICOS



ATRIBUTOS Y MÉTODOS ESTÁTICOS

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

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

ATRIBUTOS Y MÉTODOS ESTÁTICOS

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

```
Persona.hoy = “domingo”;  
System.out.println(Persona.hoy); // mostrará “domingo”
```

ATRIBUTOS Y MÉTODOS ESTÁTICOS

Un atributo estático se inicializará 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:

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

ATRIBUTOS Y MÉTODOS ESTÁTICOS

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 representa los días de la semana, de lunes a domingo.

ATRIBUTOS Y MÉTODOS ESTÁTICOS

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


ATRIBUTOS Y MÉTODOS ESTÁTICOS

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
```

ATRIBUTOS Y MÉTODOS ESTÁTICOS

Por otra parte, desde un método estático solo se puede 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. En la Actividad Resuelta 7.12 veremos un ejemplo de ello.

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 los valores numéricos primitivos y char, **null** para las referencias y **false** para los booleanos.

CONSTRUCTORES

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;
```

CONSTRUCTORES

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.

CONSTRUCTORES

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**:

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

CONSTRUCTORES

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
```

CONSTRUCTORES

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

CONSTRUCTORES

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.

CONSTRUCTORES

```
class Persona {  
    ...  
    // constructor que asigna valores a todos los atributos  
    Persona (String nombre, int edad, double estatura) {  
        this.nombre = nombre; // asigna el parámetro al atributo  
        this.edad = edad;  
        this.estatura = estatura;  
    }  
  
    // constructor sobrecargado: solo asigna el nombre  
    Persona (String nombre) {  
        this.nombre = nombre;  
        estatura = 1.0; // valor arbitrario para la estatura  
        // al no asignar la edad se inicializará por defecto: a 0  
    }  
}
```

CONSTRUCTORES

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

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

```
Persona b = new Persona("Dolores");
```

CONSTRUCTORES

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.

CONSTRUCTORES

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( );
```

CONSTRUCTORES. *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()**:

CONSTRUCTORES. this()

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

CONSTRUCTORES. *this()*

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.

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.

PAQUETES

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.

PAQUETES

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:

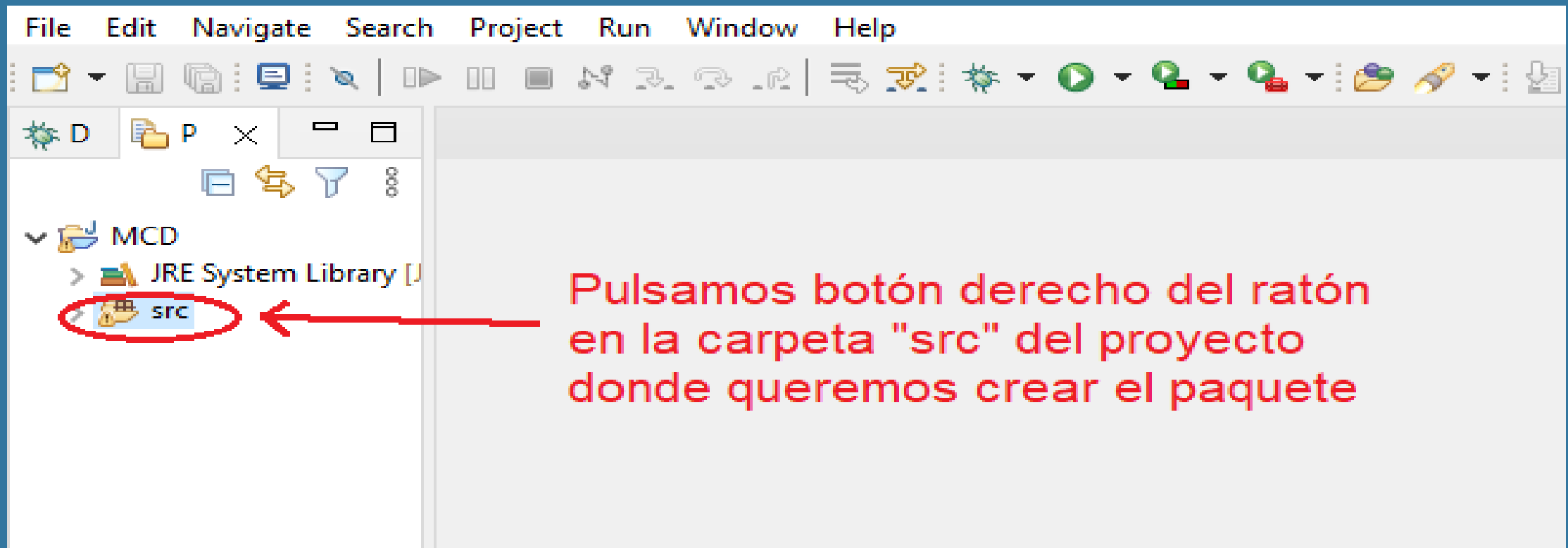
- Una sentencia donde se especifica el paquete al que pertenece, que empieza con la palabra clave **package** seguida del nombre del paquete.
- Una serie opcional de sentencias de importación, con la palabra reservada **import**, que permite importar clases definidas en otros paquetes.

PAQUETES

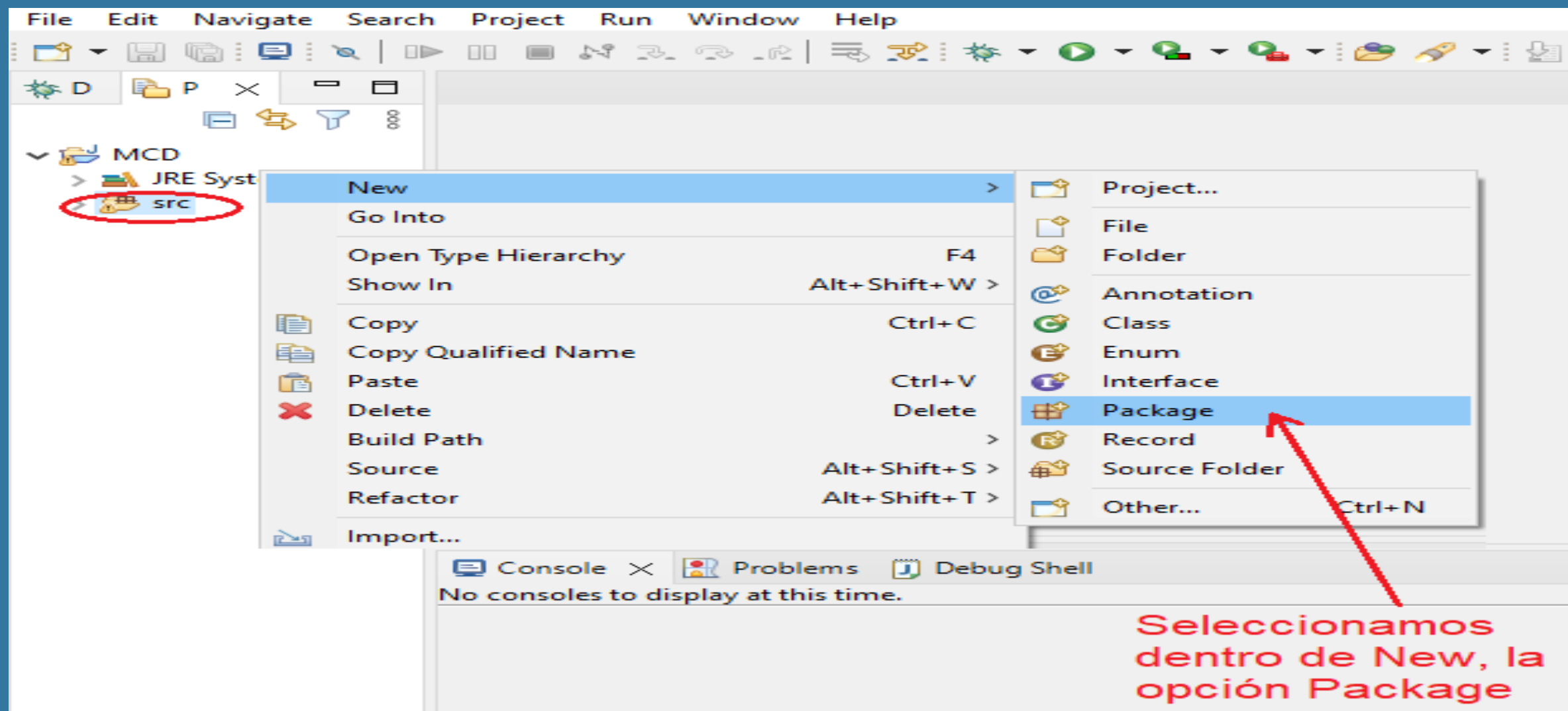
- La definición de una o más clases, de las cuales solo una puede ser declarada pública, por medio del modificador de acceso **public**. De todas formas, es recomendable que en cada archivo fuente se defina una sola clase, que debe tener el nombre del archivo.

PAQUETES. Crear un paquete en Eclipse

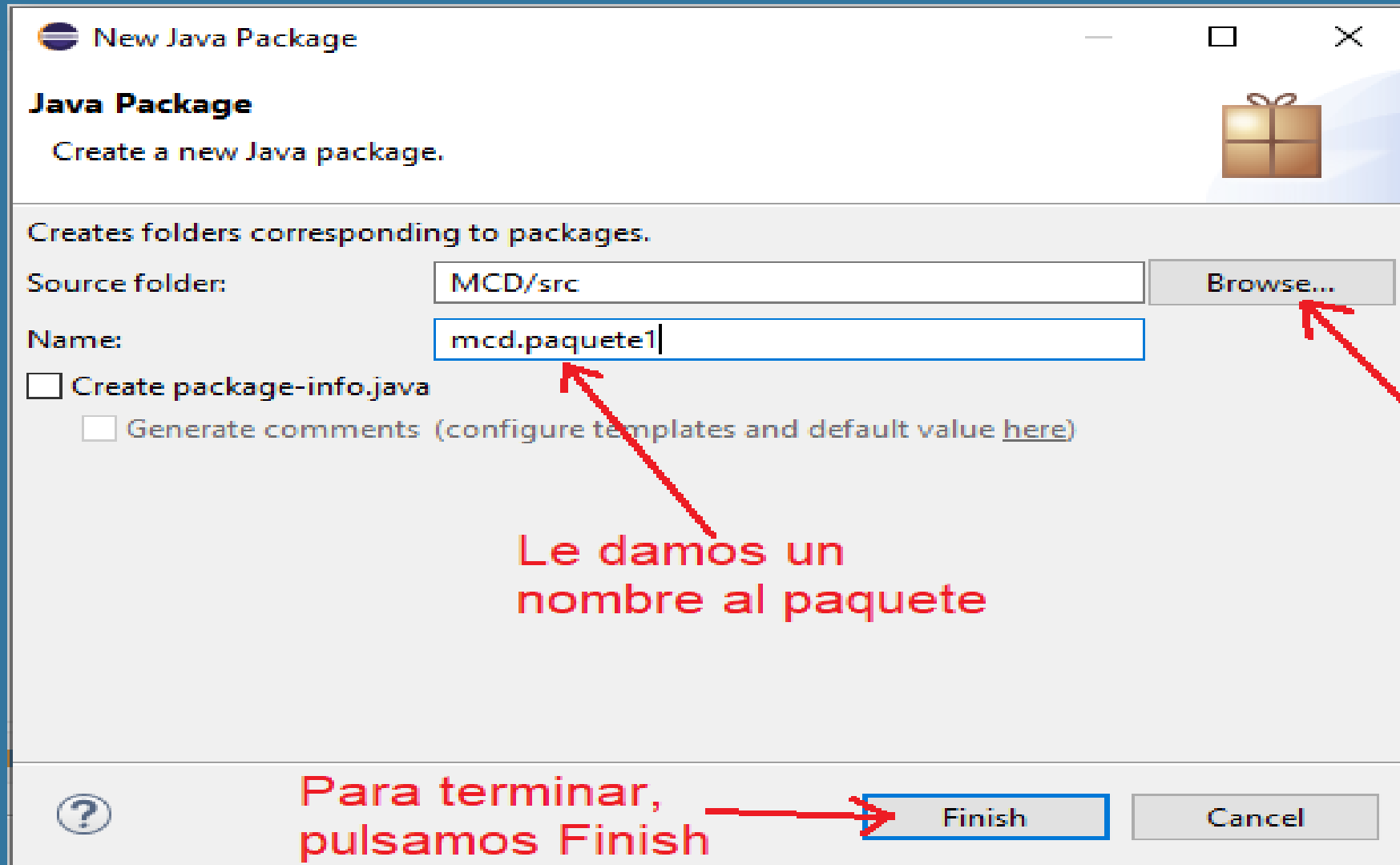
Hay distintas formas de crear un paquete en Eclipse, a continuación se expondrá una de ellas:



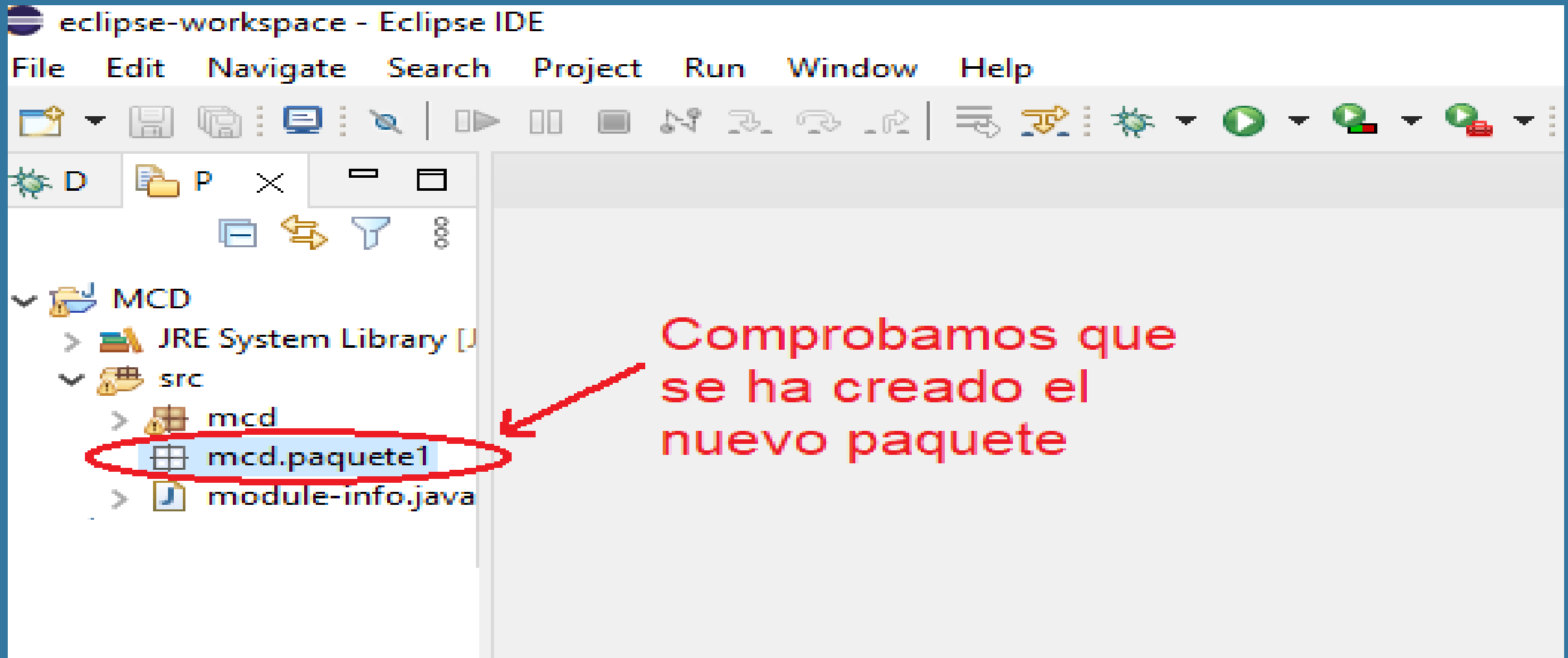
PAQUETES. Crear un paquete en Eclipse



PAQUETES. Crear un paquete en Eclipse



PAQUETES. Crear un paquete en Eclipse



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.

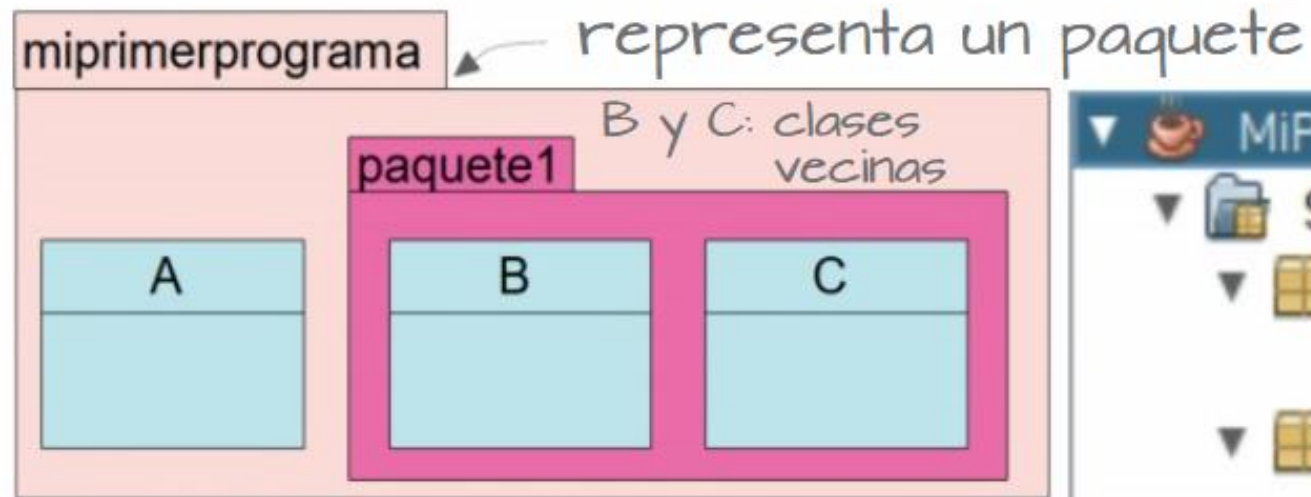
De igual manera que podemos modificar la visibilidad entre las clases, es posible modificar la visibilidad entre los miembros de distintas clases, es decir, qué atributos y métodos son visibles para otras clases.

MODIFICADORES DE ACCESO. *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 (ver la siguiente figura):

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

MODIFICADORES DE ACCESO. *Modificadores de acceso para clases*



A es externa al paquete: `miprimerprograma.paquete1` y a todas sus clases



MODIFICADORES DE ACCESO. *Modificadores de acceso para clases*

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”***.

MODIFICADORES DE ACCESO. *Modificadores de acceso para clases*

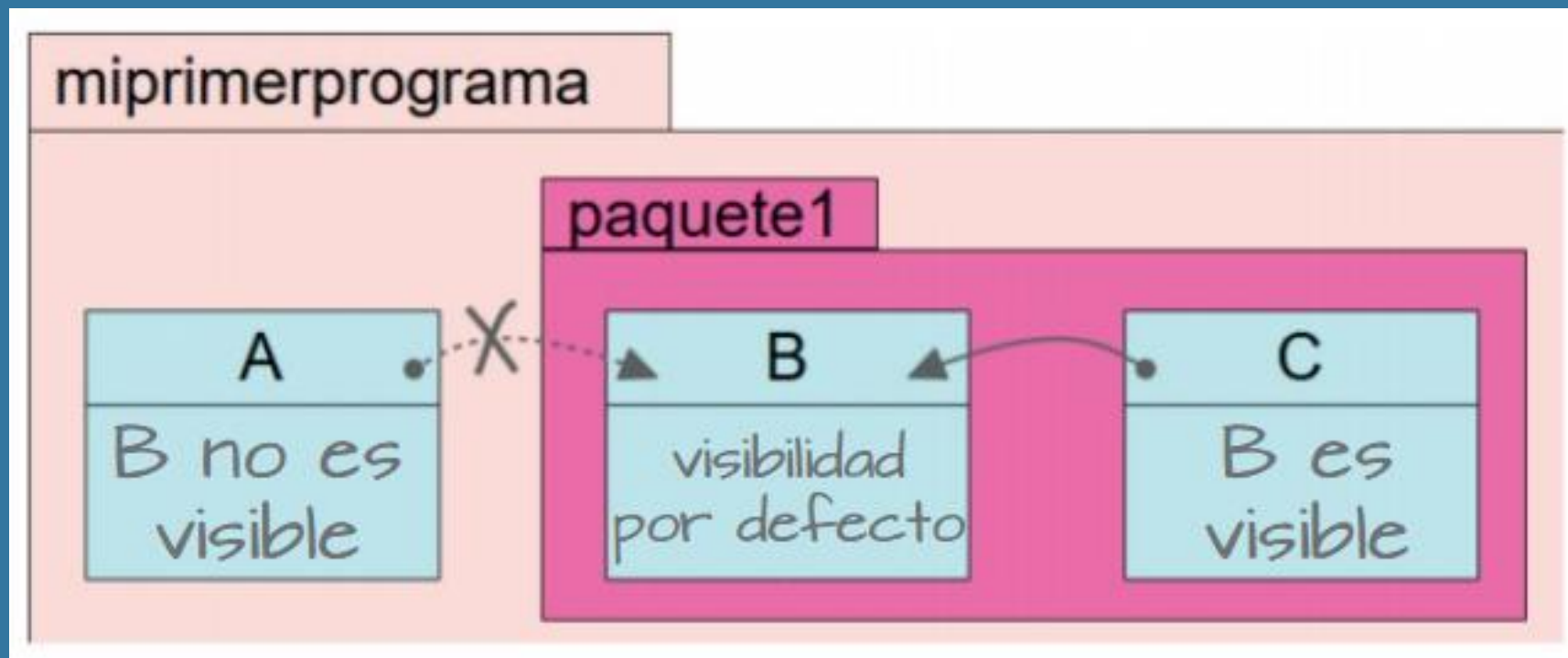
Visibilidad por defecto

Cuando definimos una clase sin utilizar ningún modificador de acceso,

```
package miprimerprograma.paquete1;  
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 (ver la siguiente figura).

MODIFICADORES DE ACCESO. *Modificadores de acceso para clases*



MODIFICADORES DE ACCESO. *Modificadores de acceso para clases*

Visibilidad total

En la figura anterior, 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.

MODIFICADORES DE ACCESO. *Modificadores de acceso para clases*

Visibilidad total

Vamos a redefinir B para que tenga visibilidad total:

```
package miprimerprograma.paquete1;  
public class B { // clase marcada como pública  
    ...  
}
```


MODIFICADORES DE ACCESO. *Modificadores de acceso para clases*

Visibilidad total

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.paquete1.B; // ahora A puede usar la
```

```
    clase B
```

```
class A {
```

```
    ...
```

```
}
```

MODIFICADORES DE ACCESO. *Modificadores de acceso para clases*

Visibilidad total




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  
miprimerprograma.paquete1  
class A {  
    ...  
}
```

MODIFICADORES DE ACCESO. *Modificadores de acceso para clases*

Visibilidad total

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

	Visible desde ...	
	Clases vecinas	Clases externas
Sin modificador		
public		

MODIFICADORES DE ACCESO. *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 de acceder a sus miembros.

MODIFICADORES DE ACCESO. *Modificadores de acceso para 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  
}
```

MODIFICADORES DE ACCESO. *Modificadores de acceso para miembros*

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.

MODIFICADORES DE ACCESO. *Modificadores de acceso para miembros*

Visibilidad por defecto

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**.

MODIFICADORES DE ACCESO. *Modificadores de acceso para miembros*

Visibilidad por defecto

No olvidemos que la clase A se ha definido como **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.

MODIFICADORES DE ACCESO. *Modificadores de acceso para miembros*

Modificador de acceso **private** y **public**

Con el modificador **private** obtenemos una visibilidad más restrictiva que por defecto, ya que impide el acceso 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.

MODIFICADORES DE ACCESO. *Modificadores de acceso para miembros*

Modificador de acceso **private** y **public**

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.

MODIFICADORES DE ACCESO. *Modificadores de acceso para miembros*

Modificador de acceso *private* y *public*

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.

MODIFICADORES DE ACCESO. *Modificadores de acceso para miembros*

Modificador de acceso private y public

```
public class Alumno {  
    public String nombre; // atributo público  
    private double notaMedia; // atributo privado  
    String dirección; // 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;  
        }  
    }  
}
```

MODIFICADORES DE ACCESO. *Modificadores de acceso para miembros*

Modificador de acceso private y public

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.

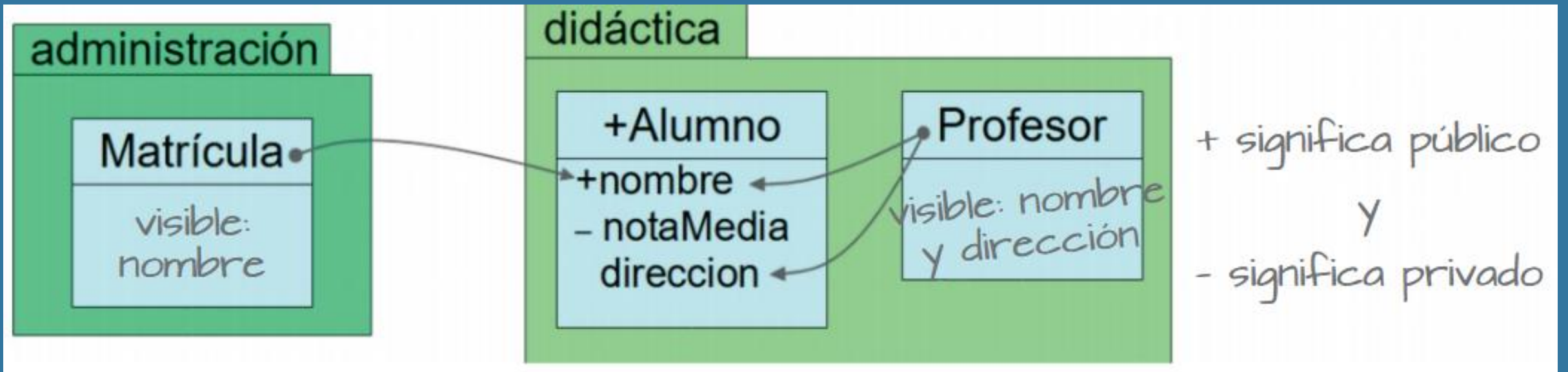
MODIFICADORES DE ACCESO. *Modificadores de acceso para miembros*

Modificador de acceso private y public

	Visible desde ...		
	La propia clase	Clases vecinas	Clases Externas
private			
sin modificador			
public			

MODIFICADORES DE ACCESO. *Modificadores de acceso para miembros*

Modificador de acceso private y public



MODIFICADORES DE ACCESO. *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.

MODIFICADORES DE ACCESO. *Métodos get/set*

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. 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.

MODIFICADORES DE ACCESO. Métodos get/set

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

```
class Persona {  
    private int edad;  
    ...  
    public void setEdad (int edad) {  
        if (edad >= 0) { // solo los valores positivos tienen sentido  
            this.edad = edad;  
        } // en caso contrario no se modifica la edad  
    }  
    public int getEdad ( ) {  
        return edad;  
    }  
}
```

MODIFICADORES DE ACCESO. *Métodos get/set*

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 de valores no negativos para la edad.

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  
}
```

ENUMERADOS

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 o bien dentro de la definición de la clase donde se va a usar. En el primer caso, en Eclipse se puede crear un tipo Enumerado, de la misma forma que creábamos una clase, pero seleccionando New -> Enum.

ENUMERADOS

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

DiaDeLaSemana ingles = DiaDeLaSemana.LUNES;

ENUMERADOS

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);
```

ENUMERADOS

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  
    ...  
}
```


ENUMERADOS

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);  
}
```

ENUMERADOS

2. Definirlo de tipo Sexo directamente.

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

ENUMERADOS

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);
```

ENUMERADOS

En el 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));
```

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

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

ENUMERADOS

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

NOTA TÉCNICA

Los ***wrappers*** o envoltorios son clases que internamente contienen un dato de tipo primitivo, lo que proporciona una forma de trabajar con estos como objetos. Cada tipo primitivo tiene su correspondiente clase envoltorio: **Integer** para el tipo `int`, **Double** para el tipo `double`, **Character** para el tipo `char`, etcétera.

ACTIVIDADES RESUELTAS

Actividad Resuelta 7.1. Diseñar la clase ***CuentaCorriente***, que almacena los datos: DNI y nombre del titular, así como el saldo. Las operaciones típicas con una cuenta corriente son:

- Crear una cuenta: se necesita el DNI y nombre del titular. El saldo inicial será 0.
- Sacar dinero: el método debe indicar si ha sido posible llevar a cabo la operación, si existe saldo suficiente.
- Ingresar dinero: se incrementa el saldo.
- Mostrar información: muestra la información disponible de la cuenta corriente.

ACTIVIDADES RESUELTAS

Actividad Resuelta 7.2. En la clase ***CuentaCorriente*** sobrecargar los constructores para poder crear objetos.

- Con el DNI del titular de la cuenta y un saldo inicial.
- Con el DNI, nombre y el saldo inicial.

Escribir un programa que compruebe el funcionamiento de los métodos.

ACTIVIDADES RESUELTAS

Actividad Resuelta 7.3. Modificar la visibilidad de la clase ***CuentaCorriente*** para que sea visible desde clases externas y la visibilidad de sus atributos para que:

- **saldo** no sea visible para otras clases.
- **nombre** sea público para cualquier clase.
- **dni** solo sea visible por clases vecinas.

Realizar un programa para comprobar la visibilidad de los atributos.

ACTIVIDADES RESUELTAS

Actividad Resuelta 7.4. Todas las cuentas corrientes con las que se va a trabajar pertenecen al mismo banco. Añadir un atributo que almacene el nombre del banco (que es único) en la clase ***CuentaCorriente***. Diseñar un método que permita recuperar y modificar el nombre del banco (al que pertenecen todas las cuentas corrientes).

ACTIVIDADES RESUELTAS

Actividad Resuelta 7.5. Existen gestores que administran las cuentas bancarias y atienden a sus propietarios.

Cada cuenta, en caso de tenerlo, cuenta con un único gestor. Diseñar la clase **Gestor** de la que interesa guardar su nombre, teléfono y el importe máximo autorizado con el que pueden operar. Con respecto a los gestores, existen las siguientes restricciones:

- Un gestor tendrá siempre un nombre y un teléfono.
- Si no se asigna, el importe máximo autorizado por operación será de 10 000 euros.
- Un gestor, una vez asignado, no podrá cambiar su número de teléfono. Y todo el mundo podrá consultarlo.

ACTIVIDADES RESUELTAS

El nombre será público y el importe máximo solo será visible por clases vecinas.

Modificar la clase ***CuentaCorriente*** para que pueda disponer de un objeto **Gestor**. Escribir los métodos necesarios.

ACTIVIDADES RESUELTAS

Actividad Resuelta 7.6. Escribir un programa que lea por teclado una hora cualquiera y un número ***n*** que representa una cantidad en segundos. El programa mostrará la hora introducida y las ***n*** siguientes, que se diferencian en un segundo. Para ello hemos de diseñar previamente la clase **Hora** que dispone de los atributos hora, minuto y segundo. Los valores de los atributos se controlan mediante métodos set/get.

ACTIVIDADES RESUELTAS

Actividad Resuelta 7.7. Diseñar la clase ***Texto*** que gestiona una cadena de caracteres con algunas características:

- La cadena de caracteres tendrá una longitud máxima que se especifica en el constructor.
- Permite añadir un carácter al principio o al final, siempre y cuando no se exceda la longitud máxima, es decir, exista espacio disponible.
- Igualmente, permite añadir una cadena, al principio o al final del texto, siempre y cuando no se rebase el tamaño establecido.

ACTIVIDADES RESUELTAS

- Es necesario saber cuántas vocales (mayúsculas y minúsculas) hay en el texto.
- Cada objeto de tipo **Texto** tiene que conocer la fecha en la que se creó, así como la fecha y hora de la última modificación efectuada.
- Deberá existir un método que muestre la información que gestiona cada texto.

ACTIVIDADES RESUELTAS

Actividad Resuelta 7.8. Definir una clase que permita controlar un sintonizador digital de emisoras FM; concretamente, se desea dotar al controlador de una interfaz que permita subir (up) o bajar (down) la frecuencia (en saltos de 0,5 MHz) y mostrar la frecuencia sintonizada en un momento dado (display). Supondremos que el rango de frecuencias para manejar oscila entre los 80 MHz y los 108 MHz y que, al inicio, el controlador sintonice la frecuencia indicada en el constructor o 80 MHz por defecto. Si durante una operación de subida o bajada se sobrepasa uno de los límites, la frecuencia sintonizada debe pasar a ser la del extremo contrario. Escribir un pequeño programa principal para probar su funcionamiento.

ACTIVIDADES RESUELTAS

Actividad Resuelta 7.9. Modelar una casa con muchas bombillas, de forma que cada bombilla se pueda encender o apagar individualmente. Para ello, hacer una clase **Bombilla** con una variable privada que indique si está encendida o apagada, así como un método que nos diga el estado de una bombilla concreta. Además, queremos poner un interruptor general, de forma que si este se apaga, todas las bombillas quedan apagadas. Cuando el interruptor general se activa, las bombillas vuelven a estar encendidas o apagadas, según estuvieran antes. Cada bombilla se enciende y se apaga individualmente, pero solo responde que está encendida si su interruptor particular está activado y además hay luz general.

ACTIVIDADES RESUELTAS

Actividad Resuelta 7.10. Hemos recibido el encargo de un cliente para definir los paquetes y clases necesarias (solo implementar los atributos y los constructores) para gestionar una empresa ferroviaria, en la que se distinguen dos grandes grupos: el personal y la maquinaria. En el primero se ubican todos los empleados de la empresa, que se clasifican en tres grupos: los maquinistas, los mecánicos y los jefes de estación. De cada uno de ellos es necesario guardar:

- **Maquinistas:** su nombre, DNI, sueldo y el rango que tienen adquirido.

ACTIVIDADES RESUELTAS

- **Mecánicos:** su nombre, teléfono (para contactar en caso de urgencia) y en qué especialidad desarrollan su trabajo (esta puede ser: frenos, hidráulica, electricidad o motor).
- **Jefes de estación:** su nombre, DNI y la fecha en la que fue nombrado jefe de estación.

En la parte de maquinaria podemos encontrar trenes, locomotoras y vagones. De cada uno de ellos hay que considerar:

- **Vagones:** tienen un número que los identifica, una carga máxima (en kilos), la carga actual y el tipo de mercancía con el que están cargados.

ACTIVIDADES RESUELTAS

- **Locomotoras:** disponen de una matrícula (que las identifica), la potencia de sus motores y una antigüedad (año de fabricación). Además, cada locomotora tiene asignado un mecánico que se encarga de su mantenimiento.
- **Trenes:** están formados por una locomotora y un máximo de 5 vagones. Cada tren tiene asignado un maquinista que es responsable de él.

ACTIVIDADES RESUELTAS

Todas las clases correspondientes al personal (**Maquinista, Mecanico y JefeEstacion**) serán de uso público. Entre las clases relativas a la maquinaria solo será posible construir, desde clases externas, objetos de tipo **Tren** y de tipo **Locomotora**. La clase **Vagon** será solo visible por sus clases vecinas.

ACTIVIDADES RESUELTAS

Actividad Resuelta 7.11. Las listas son estructuras dinámicas de datos donde se pueden insertar o eliminar elementos de un determinado tipo sin limitación de espacio.

Implementar la clase ***Lista*** correspondiente a una lista de números de la clase **Integer**. Los números se guardarán en una tabla que se redimensionará con las inserciones y eliminaciones, aumentando o disminuyendo la capacidad de la lista según el caso.

Entre los métodos de la clase, se incluirán las siguientes tareas:

- Un constructor que inicialice la tabla con un tamaño 0.

ACTIVIDADES RESUELTAS

- Obtener el número de elementos insertados en la lista.
- Insertar un número al final de la lista.
- Insertar un número al principio de la lista.
- Insertar un número en un lugar de la lista cuyo índice, que es el de la tabla, se pasa como parámetro.
- Añadir al final de la lista los elementos de otra lista que se pasa como parámetro.
- Eliminar un elemento cuyo índice en la lista se pasa como parámetro.
- Obtener el elemento cuyo índice se pasa como parámetro.
- Buscar un número en la lista, devolviendo el índice del primer lugar donde se encuentre. Si no está, devolverá -1.
- Mostrar los elementos de la lista por consola.

ACTIVIDADES RESUELTAS

Actividad Resuelta 7.12. Añadir a la clase **Lista** el método estático:

Lista concatena(Lista l1, Lista l2)

que construye y devuelve una lista que contiene, en el mismo orden, una copia de todos los elementos de l1 y l2.

ACTIVIDADES RESUELTAS

Actividad Resuelta 7.13. Una pila es una estructura dinámica de datos donde los elementos se insertan (se apilan) y se retiran (se desapilan) siguiendo la norma de que el último que se apila será el primero en desapilarse, como ocurre en una pila de platos. Cuando vamos a retirar un plato de una pila a nadie se le ocurre tirar de uno de los de abajo; retiramos (desapilamos) el que está encima de todos, que fue el último en ser apilado. Se llama ***cima*** de la pila al último elemento apilado (o al primer elemento para desapilar). Los métodos fundamentales de una pila son **apilar()** y **desapilar()**.

ACTIVIDADES RESUELTAS

Implementar la clase **Pila** para números **Integer**, donde se usa una lista (un objeto de la clase Lista implementada en la Actividad resuelta 7.11) para guardar los elementos apilados.

ACTIVIDADES RESUELTAS

Actividad Resuelta 7.14. Implementar el método no estático:

void insertarFinal(int nuevo)

que inserta un número entero al final de **tablaEnteros[]**, que es un atributo no estático de la clase **Main**. Escribir un programa que inicialice la tabla con los números del 1 al 10 y después la muestre por consola.