Tema 6-2 Clases y Objetos

1. CONCEPTOS FUNDAMENTALES DE LA POO

En este primer punto veremos una introducción a los conceptos fundamentales que intervienen en un programa orientado a objetos y más adelante entraremos en detalle en algunos de ellos.

CLASES

Una clase describe un conjunto de objetos que tienen:

- Las mismas características
- El mismo comportamiento

Las características que contiene una clase se llaman atributos, propiedades o datos de la clase.

El comportamiento son los métodos de la clase.

Podemos ver una clase como un **molde** o una **plantilla** que describe las características y el comportamiento que tienen todos los objetos que pertenecen a esa clase.

Por ejemplo, podemos pensar en una **clase Coche**.

Cuando hablamos de un **Coche** y de las características que tienen todos los coches podemos decir que todos los coches son de una marca y modelo determinados, tienen una matrícula, un color, utilizan un tipo de combustible, tienen una potencia máxima, unas dimensiones, velocidad máxima, consumo, etc., etc. No estamos pensando en **ningún coche en particular** sino en las características comunes a todos ellos.

Estas características serían los atributos o propiedades de la clase Coche.

Si pensamos en lo que podemos hacer con un coche, un coche lo podemos arrancar, detener, acelerar, frenar, repostar, etc., etc.

Estas operaciones serían los métodos de la clase Coche.

Podemos representar una clase de forma gráfica así:

	Clase Coche
Atributos	Marca Modelo Color Matricula …..
Métodos	Arrancar Detener Acelerar Frenar ….

OBJETOS

Un objeto es una instancia de una clase

Un objeto en un elemento concreto creado a partir de la clase.

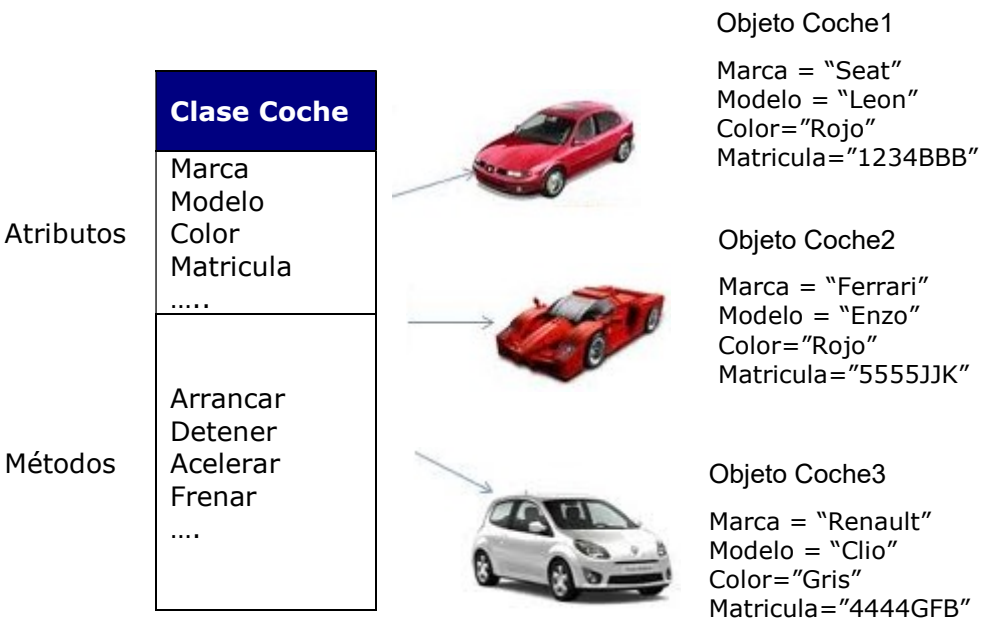
Está compuesto por un conjunto de datos (atributos) y un conjunto de operaciones que podemos realizar sobre esos datos (métodos).

Los métodos definen el comportamiento del objeto.

El estado de un objeto lo determinan los valores que toman sus datos.

A partir de una clase podemos crear distintos objetos de la misma clase.

En el ejemplo anterior la clase Coche define el molde para crear objetos de tipo coche.



MENSAJE

Un mensaje es una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos.

Cuando se ejecuta el programa los objetos están recibiendo, interpretando y respondiendo a mensajes de otros objetos. Cuando un objeto recibe un mensaje, debe conocer perfectamente lo que tiene que hacer, y cuando un objeto envía un mensaje no necesita conocer cómo se desarrolla, sino simplemente que se está desarrollando.

El conjunto de mensajes a los que un objeto puede responder se denomina **protocolo**.

Un mensaje está asociado con un método de tal forma que cuando se produce el mensaje se ejecuta el correspondiente método de la clase a la que pertenece el objeto.

El envío de un mensaje equivale a llamar a un método.

En el ejemplo anterior, un objeto coche puede recibir el mensaje de *Frenar*. En este caso se ejecutará el método correspondiente.

MÉTODO

Un método es una función asociada a un objeto, que se ejecuta tras la recepción de un mensaje.

El conjunto de métodos definidos en la clase determinan el comportamiento del objeto, es lo que el objeto puede hacer.

Un método puede producir un cambio en las propiedades del objeto o generar un evento con un nuevo mensaje para otro objeto del sistema.

2. CLASES

La clase es la unidad fundamental de programación en Java.

Un programa Java está formado por un conjunto de clases.

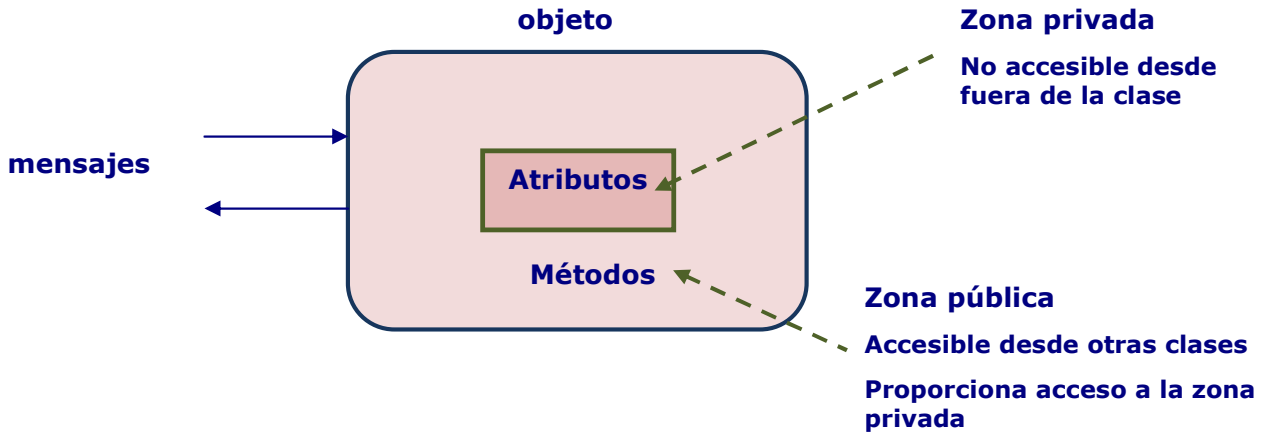
A partir de esas clases se crearán los objetos.

Una **clase** representa al conjunto de objetos que comparten unas características y un comportamiento comunes.

Puede considerarse como una **plantilla o prototipo para crear objetos**: define los atributos que componen ese tipo de objetos y los métodos que pueden emplearse para trabajar con esos objetos.

Los atributos definen el estado de cada objeto de esa clase y los métodos su comportamiento.

Según el principio de encapsulamiento, que es una de las características fundamentales de la POO, los atributos debemos considerarlos como la zona más interna o zona privada. Esta zona interna debe estar oculta a los usuarios de los objetos de la clase. El acceso a esta zona se realizará a través de los métodos.



Sintaxis general para definir una clase en Java:

```
[modificadorDeAcceso] class NombreClase [extends NombreSuperClase]
                                        [implements Interface1, Interface2, … ] {


     //atributos de la clase (cero ó más atributos)
     [modificadorDeAcceso]  tipo nombreAtributo;


     //métodos de la clase (cero ó más métodos)
     [modificadorDeAcceso] tipo nombreMetodo([parámetros]) [throws listaExcepciones]{
                               // instrucciones del método
                               [return [valor];]
     }
} //final de la clase
```

Todo lo que aparece entre corchetes es opcional. Una clase puede tener cero o más atributos y cero o más metodos, por lo tanto la definición mínima de una clase es:

```
class NombreClase{
}
```

Modificador de acceso: El concepto de clase incluye la idea de **ocultación de datos**, que básicamente consiste en que no se puede acceder a los datos directamente (zona privada), sino que hay que hacerlo a través de los métodos de la clase.

De esta forma se consiguen dos objetivos importantes:

- Que el usuario no tenga acceso directo a la estructura interna de la clase, para no poder generar código basado en la estructura de los datos.
- Si en un momento dado alteramos la estructura de la clase todo el código del usuario no tendrá que ser retocado.

El **modificador de acceso se utiliza para definir el nivel de ocultación o visibilidad** de los miembros de la clase (atributos y métodos) y de la propia clase.

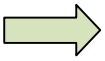
Los modificadores de acceso **ordenados de menor a mayor visibilidad** son:

MODIFICADOR DE ACCESO	EFECTO	APLICABLE A
private	Restringe la visibilidad al **interior de la clase**. Un atributo o método definido como *private* solo puede ser usado en el interior de su propia clase.	Atributos Métodos
<Sin modificador>	Cuando no se especifica un modificador, el elemento adquiere el *acceso por defecto o friendly*. También se le conoce como acceso de package (paquete). Solo puede ser usado por las **clases dentro de su mismo paquete**.	Clases Atributos Métodos
protected	Se emplea en la herencia. El elemento puede ser utilizado por **cualquier clase dentro de su paquete y por cualquier subclase** independientemente del paquete donde se encuentre.	Atributos Métodos
public	Es el nivel máximo de visibilidad. El elemento es visible desde cualquier clase.	Clases Atributos Métodos

class: palabra reservada para crear una clase en Java.

NombreClase: nombre de la clase que se define. Si la clase es pública, el nombre del archivo que la contiene debe coincidir con este nombre. El nombre debe describir de forma apropiada la entidad que se quiere representar. Los nombres deben empezar por mayúscula y si está formado por varias palabras, la primera letra de cada palabra irá en mayúsculas.

extends *NombreSuperclase*: extends es la palabra reservada para indicar la **herencia** en Java. *NombreSuperClase* es la clase de la que hereda esta clase. Si no aparece extends la clase hereda directamente de una clase general del sistema llamada **Object.**



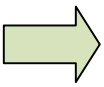
Object es la raíz de toda la jerarquía de clases en Java. Es la superclase de las que heredan directa o indirectamente todas las clases Java.

Cuando una clase hereda de otra, se llama **superclase** o **clase base** a la clase de la que hereda la nueva clase, llamada **clase derivada** o **subclase**. **La clase derivada hereda todos los atributos y métodos de su superclase**.

implements *NombreInterface1*, *NombreInterface2*, ... : implements es la palabra reservada para indicar que la clase implementa la o las interfaces que se indican separados por comas.

Importante

En Java solo puede haber una clase pública por archivo de código fuente .java



El nombre de la clase pública debe coincidir con el nombre del archivo fuente. Por ejemplo, si el nombre de la clase pública es *Persona* el archivo será *Persona.java*

En una aplicación habrá una clase principal que será la que contenga el método *main*. Esta clase deberá haber sido declarada como pública

Junto al modificador de acceso pueden aparecer **otros modificadores** aplicables a clases, atributos y métodos:

MODIFICADOR	EFECTO	APLICABLE A
abstract	Aplicado a una clase, la declara como clase abstracta. No se pueden crear objetos de una clase abstracta. Solo puede usarse como superclase. Aplicado a un método, la definición del método se hace en las subclases.	Clases Métodos
final	Aplicado a una clase significa que no se puede extender (heredar), es decir que no puede tener subclases. Aplicado a un método significa que no puede ser sobrescrito en las subclases. Aplicado a un atributo significa que contiene un valor constante que no se puede modificar	Clases Atributos Métodos
static	Aplicado a un atributo indica que es una variable de clase. Esta variable es única y compartida por todos los objetos de la clase. Aplicado a un método indica que se puede invocar sin crear ningún objeto de su clase.	Atributos Métodos
volatile	Un atributo volatile puede ser modificado por métodos no sincronizados en un entorno multihilo.	Atributos
transient	Un atributo transient no es parte del estado persistente del objeto.	Atributos
sincronized	Métodos para entornos multihilo.	Métodos

Como ejemplo, vamos a definir una clase Persona.

```
public class Persona {

    private String nombre;
    private int edad;

    public void setNombre(String nom) {
        nombre = nom;
    }

    public String getNombre() {
        return nombre;
    }

    public void setEdad(int ed) {
        edad = ed;
    }

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

ATRIBUTOS ó DATOS DE LA CASE

MÉTODOS

La clase Persona es pública y tiene dos atributos privados, nombre y edad, y cuatro métodos públicos.

Los métodos públicos que aparecen en esta clase se conocen como **métodos de acceso ó métodos setters/getters**.

Son métodos que sirven para asignar y obtener los valores de los atributos privados de la clase.

En cada clase es conveniente escribir un método set y otro get para cada atributo privado para poder acceder al contenido de los atributos desde fuera de la clase.

Recuerda que por el principio de encapsulamiento, los atributos de la clase son privados y por lo tanto inaccesibles desde fuera de la clase. Por lo tanto se necesita un método público *get* y otro método público *set* para poder acceder al contenido de los atributos desde fuera de la clase.

3. MIEMBROS DE UNA CLASE

ATRIBUTOS

Una clase puede tener **cero o más atributos**.

Sirven para almacenar los datos de los objetos. En el ejemplo anterior almacenan el nombre y la edad de cada objeto Persona.

Se declaran generalmente al principio de la clase.

La declaración es similar a la declaración de una variable local en un método.

La declaración contiene un modificador de acceso de los vistos anteriormente: *private*, *package*, *protected*, *public*.

Pueden ser variables de tipo primitivo o referencias a objetos.

A diferencia de las variables locales que se declaran dentro de un método, los atributos de una clase toman un valor inicial por defecto:

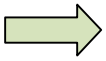
- 0 para tipos numéricos
- '\0' para el tipo char
- null para String y resto de referencias a objetos.

También se les puede asignar un valor inicial en la declaración aunque lo normal es hacerlo en el constructor.

En la clase Persona se ha declarado edad de tipo primitivo y nombre de tipo String. Ambas private y por lo tanto solo accesibles desde los métodos de la propia clase.

```
private String nombre;
private int edad;
```

El valor inicial de nombre es null y el de edad es 0.


El valor de los atributos en cada momento determina lo que se conoce como **estado** del objeto.

Dentro de una clase puede haber dos **tipos de atributos**:

- **Atributos de instancia**: son todos los atributos que no se han declarado como **static**. Son atributos específicos para cada objeto. Cada objeto de la clase tiene sus propios valores para estas variables, es decir, **cada objeto que se crea contendrá su propia copia de los atributos con sus propios valores**.
- **Atributos de clase**: son los declarados **static**. También se llaman **atributos estáticos**. Un atributo de clase no es específico de cada objeto. **Solo hay una copia del mismo y su valor es compartido por todos los objetos de la clase.** Un atributo de clase existe y puede utilizarse aunque no existan objetos de la clase. Podemos considerarlo como una ***variable global*** a la que tienen acceso todos los objetos de la clase.

  Para acceder a un atributo de clase se escribe: *NombreClase.Atributo;*

  Por ejemplo, en la clase Persona podemos añadir un atributo *contadorPersonas* que indique cuantos objetos de la clase se han creado. Sería un atributo de clase ya que no es un valor propio de cada persona. Este atributo se declara así:

  ```
  static int contadorPersonas;
  ```

  Cada vez que se crea una persona podemos incrementar su valor:

  ```
  Personas.contadorPersonas++;
  ```

Si lo declaramos además como private:

```
private static int contadorPersonas
```

solo podremos acceder al atributo desde fuera de la clase a través de un método, como veremos a continuación.

MÉTODOS

Una clase puede contener **cero o más métodos**.

Los métodos definen el comportamiento de los objetos de la clase.

A través de los métodos se accede a los datos privados de la clase.

Desde el punto de vista de la POO **el conjunto de métodos de la clase se corresponden con el conjunto de mensajes a los que los objetos de esa clase pueden responder**.

Al conjunto de métodos de una clase se le llama **interfaz de la clase**.

Los métodos pueden clasificarse en:

- **Métodos de instancia**: Son todos los métodos no declarados **static**. Pueden operar con todos los atributos de la clase, tanto los atributos static como los no static.

  La sintaxis de llamada a un método de instancia es:

  ```
  objeto.metodo([parámetros]); // Llamada típica a un método de instancia
  ```

  Todas las instancias de una clase comparten la misma implementación para un método de instancia.

- **Métodos de clase**: Son los métodos declarados como **static**. También llamados métodos estáticos. Tienen acceso solo a los atributos estáticos de la clase. No es necesario instanciar (crear) un objeto para poder utilizar un método estático.

  Para acceder a un método estático se utiliza el nombre de la clase en lugar de utilizar un objeto determinado: `NombreClase.metodoEstático([parámetros]);`

  Siguiendo con el ejemplo anterior, si el atributo estático *contadorPersonas* se declara como *private* tendremos que crear dentro de la clase un método público para poder acceder a él y realizar el incremento:

  ```
  public static void incrementarContador(){
          contadorPersonas++;
  }
  ```

  Para invocar a este método desde fuera de la clase escribiremos la instrucción:

  ```
  Persona.incrementarContador();
  ```

  La API de Java proporciona muchas clases con métodos estáticos, por ejemplo, los métodos de la clase Math: Math.sqrt(), Math.pow(), etc.

4. OBJETOS

Un objeto es una **instancia** de una clase.

El nombre del objeto, también llamado referencia del objeto, almacena su dirección de memoria.

Para crear un objeto se deben realizar dos operaciones:

1. Declaración
2. Instanciación

1. Declaración

En la declaración **se crea la referencia al objeto**, de forma similar a cómo se declara una variable de un tipo primitivo. La referencia se utiliza para manejar el objeto.

La sintaxis general es:

```
NombreClase referenciaObjeto;
```

Por ejemplo, para crear un objeto de la clase Persona el primer paso será crear su referencia así:

```
Persona p;
```

La referencia tiene como misión **almacenar la dirección de memoria del objeto**. En este momento la referencia *p* almacena una dirección de memoria nula (null) ya que aun no se ha creado el objeto.

2. Instanciación

Mediante la instanciación del objeto se **reserva un bloque de memoria para almacenar todos los atributos del objeto**. Instanciar un objeto significa crear el objeto.

Solo se reserva memoria para los atributos no declarados como static.

En ese bloque de memoria no se guardan los métodos para cada objeto. El código de los métodos es el mismo y lo comparten todos los objetos de la clase.

La dirección del bloque de memoria donde se encuentra el objeto se asigna a la referencia indicada en la declaración del objeto.

De forma general un objeto se instancia de esta forma:

```
referenciaObjeto = new NombreClase();
```

new es el operador para crear objetos.

Mediante new se asigna la memoria necesaria para ubicar los atributos del objeto y devuelve la dirección de memoria donde empieza el bloque asignado.

Por ejemplo:

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

En esta instrucción, mediante el operador *new* se ha asignado un bloque de memoria para guardar los atributos de un objeto de tipo Persona (nombre y edad). La dirección de memoria del objeto se asigna a la variable *p*.

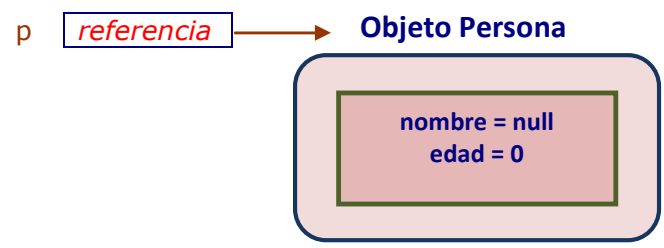
Las operaciones de declaración e instanciación pueden realizarse en la misma línea de código:

```
NombreClase referenciaObjeto = new NombreClase();
```

Por ejemplo:

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

Se ha creado un objeto de tipo Persona y se ha asignado su dirección a la variable p.



Manejar objetos

Una vez creado el objeto, lo manejamos a través de su referencia.

En general, **el acceso a los componentes del objeto se realiza a través del operador *punto* seguido del método al que queremos acceder**.

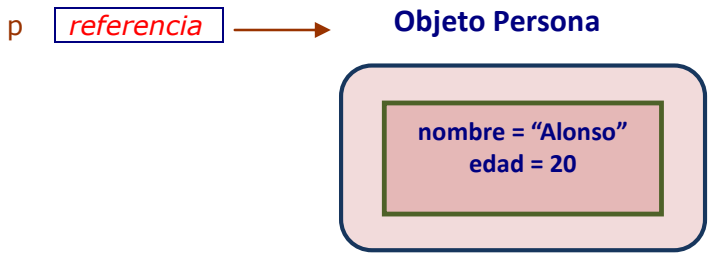
```
referenciaObjeto.método([parámetros]);
```

Por ejemplo:

Para asignar valores a los atributos *nombre* y *edad* del objeto *p* lo haremos a través de los métodos *set* ya que nombre y edad son privados y no podemos acceder directamente a ellos desde fuera de la clase:

```
p.setNombre("Alonso");
p.setEdad(20);
```

Después de estas instrucciones el objeto p se ha modificado de esta forma:



Si desde fuera de la clase Persona intentamos asignar directamente un valor a uno de los atributos privados del objeto, por ejemplo:

```
p.edad = 20;
```

el compilador nos avisará del error ya que intentamos acceder de forma directa a un atributo privado.

La utilización del modificador **private** sirve para implementar una de las características de la programación orientada a objetos: el ocultamiento de la información o **encapsulación**.

La declaración de un atributo de una clase como público no respeta este principio de ocultación de información. Declarándolos como privados, no se tiene acceso directo a los atributos del objeto fuera del código de la clase correspondiente y sólo puede accederse a ellos de forma indirecta a través de los métodos proporcionados por la propia clase.

Una de las ventajas de obligar al empleo de un método para modificar el valor de un atributo es asegurar la consistencia de los datos que contiene ese atributo. Por ejemplo, el método *setEdad* de la clase Persona lo podemos escribir para evitar que se asignen valores no permitidos para el atributo edad:

```
public void setEdad(int e) {
        if(e > 0){
           edad = e;
        }else{
              edad = 0;
        }
}
```

Con este método, si se escribe una instrucción como esta:

```
p.setEdad(-20);
```

nos aseguramos que no se asignará a *edad* un valor no válido.

Si el atributo *edad* fuese público podemos escribir desde cualquier sitio instrucciones como esta:

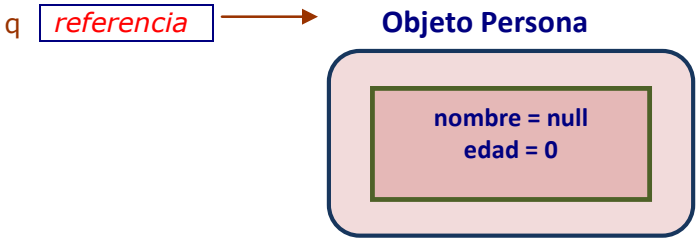
```
p.edad = -20;
```

provocando que *edad* contenga un valor no válido.

Manejar varios objetos

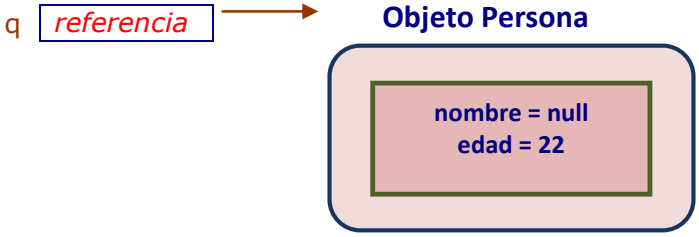
En una aplicación se pueden crear tantos objetos como sean necesarios. Por ejemplo podemos crear otro objeto de tipo Persona:

```
Persona q = new Persona();  //Se crea otro objeto persona
```



Asignamos a la edad de q el valor 22:

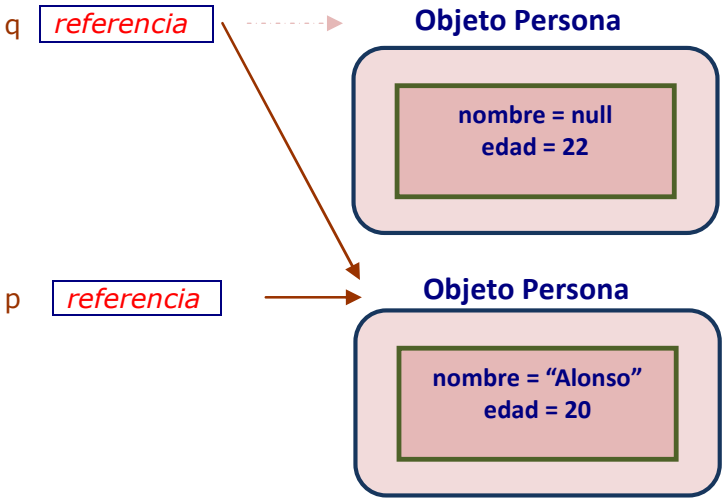
```
q.setEdad(22);
```



Si tenemos dos objetos *p* y *q*, en una asignación como esta:

```
q = p;
```

no se copian los valores de los atributos del objeto *p* al objeto *q*, sino que estamos copiando el contenido de la variable *p* a la variable *q*. La variable *p* contiene la dirección de memoria donde se encuentra el objeto por lo que ahora *q* también apunta al mismo objeto que *p*.



El objeto referenciado previamente por *q* se queda ahora sin referencia (inaccesible).

El **recolector de basura** de Java se encarga de eliminar de memoria los objetos cuando detecta que no se podrán usar más, es decir, cuando detecta que ninguna variable tiene la dirección del objeto y por lo tanto está inaccesible.

Ejemplo: Vamos a diseñar una clase para manejar fechas. La clase contendrá como atributos el día, mes y año. Los métodos de la clase serán los métodos get/set, un método para asignar la fecha completa (día, mes y año al mismo tiempo) y un método para comprobar si los valores del día, mes y año corresponden a una fecha correcta.

La definición de la clase con sus métodos podría ser esta:

```
public class Fecha {
    private int dia;
    private int mes;
    private int año;
    public void setDia(int d) {
        dia = d;
    }
    public void setMes(int m) {
        mes = m;
    }
    public void setAño(int a) {
        año = a;
    }
    public int getDia() {
        return dia;
    }
    public int getMes() {
        return mes;
    }
    public int getAño() {
        return año;
    }
    public void asignarFecha(int d, int m, int a) {
        setDia(d);
        setMes(m);
        setAño(a);
    }
    public boolean fechaCorrecta() {
        boolean diaCorrecto, mesCorrecto, anyoCorrecto;
        anyoCorrecto = (año > 0);
        mesCorrecto = (mes >= 1) && (mes <= 12);
        switch (mes) {
            case 2: if (esBisiesto()) {
                        diaCorrecto = (dia >= 1 && dia <= 29);
                    } else {
                        diaCorrecto = (dia >= 1 && dia <= 28);
                    }
                    break;
            case 4:
            case 6:
            case 9:
            case 11: diaCorrecto = (dia >= 1 && dia <= 30);
                     break;
            default: diaCorrecto = (dia >= 1 && dia <= 31);
        }
        return diaCorrecto && mesCorrecto && anyoCorrecto;
    }
    private boolean esBisiesto() {
        return ((año % 4 == 0) && (año % 100 != 0) || (año % 400 == 0));
    }
}
```

Se ha incluido en la clase el método *esBisiesto* como privado. Este método lo invoca el método *fechaCorrecta* para comprobar si el año es bisiesto o no.

Vamos ahora a utilizar la clase Fecha que hemos creado. En el método *main* de la clase principal vamos a crear un objeto de tipo Fecha y vamos a introducir por teclado los valores para los atributos día, mes y año. Si la fecha introducida no es correcta se vuelve a pedir hasta que sea correcta.

Finalmente se muestra por pantalla los valores día, mes y año asignados al objeto.

```
public class Principal {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        Fecha f = new Fecha(); //--> Se crea un objeto de tipo Fecha
        int d, m, a;
        boolean ok;
        do {
            ok = true;
            System.out.println("Introduce fecha: "); //Se introduce una fecha por teclado
            System.out.print("dia: ");
            d = sc.nextInt();
            System.out.print("mes: ");
            m = sc.nextInt();
            System.out.print("año: ");
            a = sc.nextInt();
            f.asignarFecha(d, m, a); //se asigna la fecha introducida al objeto
            if (!f.fechaCorrecta()) { //se comprueba si la fecha es válida
                System.out.println("Fecha no válida");
                ok = false;
            }
        } while (!ok); //si la fecha no es válida se vuelve a pedir
        System.out.println(f.getDia() + "/" + f.getMes() + "/" + f.getAño());
    }
}
```

5. CONSTRUCTORES

Un **constructor** es un **método especial que se invoca siempre que se crea un objeto**.

Su función es inicializar el objeto y sirve para asegurarnos que los objetos siempre contengan valores válidos.

Cuando se crea un objeto se realizan las siguientes operaciones **de forma automática**:

1. Se asigna memoria para el objeto.
2. Se invoca al constructor para inicializar los atributos del objeto.

Cada vez que se instancia un objeto, lo que se escribe a continuación del operador *new* es la llamada a un constructor de la clase.

Por ejemplo: `Fecha f = new Fecha(); //invocamos al constructor Fecha()`

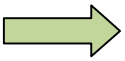
Los constructores tienen las siguientes características:

- Tiene el mismo nombre que la clase a la que pertenece.
- Una clase puede contener varios constructores, tantos como necesitemos. Esto quiere decir que los constructores se pueden sobrecargar: puede haber varios constructores con el mismo nombre y distinto número de argumentos.
- Los constructores no se heredan.
- No devuelve ningún valor ni siquiera void.
- Debe declararse *public* (salvo casos excepcionales) para que pueda ser invocado desde cualquier parte donde se desee crear un objeto de su clase.

CONSTRUCTOR POR DEFECTO

Los constructores son métodos que el programador escribe dentro de la clase.

Si en una clase no se define ningún constructor, Java proporciona automáticamente uno por defecto.



El **constructor por defecto es un constructor sin parámetros**. Los atributos del objeto que se está creando son inicializados con los valores predeterminados por el sistema.

Por ejemplo, en la clase Fecha no se ha definido ningún constructor, por lo que al crear un objeto se ha utilizado el constructor por defecto que proporciona Java.

```
Fecha f = new Fecha(); //invocamos al constructor Fecha()
```

En este caso el constructor que se ha utilizado es el constructor por defecto proporcionado por java:

```
public Fecha() {
}
```

Vemos que el constructor por defecto proporcionado por Java es un método sin parámetros y sin código. Como en el resto de constructores, tampoco se indica el tipo devuelto (que sería void). En este caso cuando se crea el objeto Fecha este constructor es el que se encarga de asignarle a sus atributos los valores por defecto.

VARIOS CONSTRUCTORES EN UNA CLASE

Una clase puede tener tantos constructores como necesitemos, según la forma en que nos interese crear los objetos.

Como ejemplo, vamos a añadir un constructor a la clase Fecha. Si vemos el código de la clase Fecha podemos comprobar que no contiene ningún constructor por lo que estamos utilizando para crear objetos el constructor por defecto que proporciona Java.

El nuevo constructor que vamos a escribir es un constructor con parámetros. El número de parámetros serán tres de tipo int correspondientes a los valores del día, mes y año.

Si utilizamos este constructor, cuando se crea un objeto de tipo Fecha al mismo tiempo que se crea se pueden inicializar sus atributos con los valores que se reciben en los parámetros. Dentro del constructor vamos a escribir el código de forma que si los valores que se le envían al constructor corresponden a una fecha incorrecta se asigna al objeto la fecha del sistema.

```
//Constructor con parámetros
public Fecha(int d, int m, int a) {
       dia = d; //se asigna a los atributos de la clase los valores recibidos
       mes = m;
       año = a;
       if (!fechaCorrecta()) { //si la fecha no es correcta se toma la fecha del sistema
            LocalDate fechaActual = LocalDate.now();
            dia = fechaActual.getDayOfMonth();
            mes = fechaActual.getMonthValue();
            año = fechaActual.getYear();
       }
}
```

Con este constructor ahora podremos crear objetos de la siguiente manera:

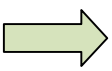
```
Fecha fecha1 = new Fecha(10,2,2010);
```

La instrucción anterior crea un objeto *fecha1* y se le asignan esos valores a sus atributos.

Si se crea un objeto con una fecha no válida:

```
Fecha fecha2 = new Fecha(10,20,2010);
```

A los atributos día, mes y año del objeto *fecha2* se les asignará los valores del sistema.



Importante: Java proporciona un constructor por defecto si no se ha definido ninguno en la clase, pero si en una clase definimos un constructor ya no tendremos disponible el constructor por defecto, por lo tanto si queremos usarlo tendremos que escribirlo expresamente dentro de la clase.

En este ejemplo hemos definido en la clase Fecha un constructor con parámetros. Ahora es el único constructor que tiene la clase ya que el constructor por defecto deja de estar disponible. Si queremos seguir utilizando el constructor por defecto tendremos que escribirlo dentro de la clase.

En la clase Fecha tendremos que añadir el constructor por defecto:

```
public Fecha(){
}
```

Este constructor lo usaremos cuando queramos crear un objeto Fecha sin asignarle unos valores determinados. El objeto se crea tomando los valores por defecto. Después durante la ejecución del programa asignaremos valores a los atributos mediante los métodos set o mediante el método *asignarFecha*.

El constructor por defecto también puede contener código. Por ejemplo, cuando se crea un objeto Fecha con el constructor por defecto podemos hacer que se asigne a sus atributos la fecha del sistema. En este caso, el constructor por defecto se escribiría así:

```
//Constructor por defecto
public Fecha() {
        LocalDate fechaActual = LocalDate.now();
        dia = fechaActual.getDayOfMonth();
        mes = fechaActual.getMonthValue();
        año = fechaActual.getYear();
}
```

La clase tiene ahora dos constructores por lo que podemos crear objetos Fecha de dos formas:

```
Fecha f1 = new Fecha();    //se ejecuta el constructor por defecto
Fecha f2 = new Fecha(1,1,2010); //se ejecuta el constructor con parámetros
```

INVOCAR A UN CONSTRUCTOR

Un constructor **se invoca** cuando se crea un objeto de la clase **mediante el operador new**.

Si fuera necesario invocarlo en otras situaciones, la llamada a un constructor solo puede realizarse desde dentro de otro constructor de su misma clase y debe ser siempre la primera instrucción. Para ello se usa **this**.

Por ejemplo, si desde el constructor con parámetros de la clase Fecha fuese necesario llamar al constructor sin parámetros (constructor por defecto) de la clase escribimos:

```
public Fecha(int d, int m, int a) {
        this();  //llamada al constructor sin parámetros.
        dia = d; //resto de código del constructor
        . . . . .
}
```

La llamada al constructor se realiza mediante la instrucción *this()*. Esta instrucción debe ser la primera que aparezca y a continuación se escribe el resto de código del constructor.

CONSTRUCTOR COPIA

Se puede crear un objeto como copia de otro de su misma clase utilizando un constructor llamado **constructor copia**.

Este constructor crea un nuevo objeto y copia los valores de los atributos de un objeto existente a los del objeto que se está creando.

Los constructores copia tiene un solo argumento, el cual es una referencia a un objeto de la misma clase que será desde el que queremos copiar.

Por ejemplo, para la clase Fecha podemos escribir un constructor copia que permita crear objetos con los valores de otro ya existente:

```
//constructor copia de la clase Fecha
public Fecha(Fecha f) {
        dia = f.dia;
        mes = f.mes;
        año = f.año;
}
```

Si creamos un objeto fecha:

```
Fecha fecha = new Fecha(1,2,2020);
```

Podemos crear otro objeto fecha1 como copia de fecha:

```
Fecha fecha1 = new Fecha(fecha);  //se invoca al constructor copia
```

El constructor copia recibe el objeto a copiar y asigna los valores de sus atributos, uno a uno, a cada atributo del objeto que estamos creando.

La clase Fecha completa con todos los constructores que hemos creado quedaría así:

```
public class Fecha {
    private int dia;
    private int mes;
    private int año;

    //Constructor por defecto.
    public Fecha() {
    }

    //Constructor con parámetros
    public Fecha(int d, int m, int a) {
       dia = d; //se asigna a los atributos de la clase los valores recibidos
       mes = m;
       año = a;
       if (!fechaCorrecta()) { //si la fecha no es correcta se toma la fecha del sistema
            LocalDate fechaActual = LocalDate.now();
            dia = fechaActual.getDayOfMonth();
            mes = fechaActual.getMonthValue();
            año = fechaActual.getYear();
       }
    }
    //Constructor copia
    public Fecha(Fecha f) {
        dia = f.dia;
        mes = f.mes;
        año = f.año;
    }
    //métodos setters/getters
    public void setDia(int d) {
        dia = d;
    }
    public void setMes(int m) {
        mes = m;
    }
```

```
    public void setAño(int a) {
        año = a;
    }
    public int getDia() {
        return dia;
    }
    public int getMes() {
        return mes;
    }
    public int getAño() {
        return año;
    }
    //método para asignar valores a los tres atributos
    public void asignarFecha(int d, int m, int a) {
        setDia(d);
        setMes(m);
        setAño(a);
    }
    //método para comprobar si la fecha es correcta
    public boolean fechaCorrecta() {
        boolean diaCorrecto, mesCorrecto, anyoCorrecto;
        anyoCorrecto = (año > 0);
        mesCorrecto = (mes >= 1) && (mes <= 12);
        switch (mes) {
            case 2:  if (esBisiesto()) {
                         diaCorrecto = (dia >= 1 && dia <= 29);
                     } else {
                         diaCorrecto = (dia >= 1 && dia <= 28);
                     }
                     break;
            case 4:
            case 6:
            case 9:
            case 11: diaCorrecto = (dia >= 1 && dia <= 30);
                     break;
            default: diaCorrecto = (dia >= 1 && dia <= 31);
        }
        return diaCorrecto && mesCorrecto && anyoCorrecto;
    }
    //método privado no disponible desde fuera de la clase.
    //Lo invoca el método fechaCorrecta para comprobar si el año es bisiesto o no
    private boolean esBisiesto() {
        return ((año % 4 == 0) && (año % 100 != 0) || (año % 400 == 0));
    }
}
```

En lugar del constructor por defecto sin código podríamos haber escrito en la clase el constructor por defecto con código que asigna a los atributos la fecha del sistema pero debemos tener en cuenta que no podemos tener dos constructores por defecto en la clase, escribiremos uno u otro según nos interese.

6. REFERENCIA THIS

Cada objeto de una determinada clase contiene su propia copia de sus datos, pero no de los métodos, de los que solo existe una única copia para todos los objetos de esa clase.

Cada objeto tiene su propia estructura de datos, pero todos los objetos de la clase comparten el código único de los métodos de la clase.

Para que un método conozca qué objeto de todos los creados lo ha llamado, Java proporciona de forma automática una referencia al objeto con el que se está trabajando llamada **this**.

Todos los métodos de una clase tienen a su disposición una **referencia this que contiene la dirección del objeto que ha invocado al método**.

Cuando se invoca un método se realizan **de forma automática** las siguientes operaciones:

- se crea la referencia this
- se le asigna la dirección de memoria del objeto que invoca al método
- se envía al método como parámetro oculto.

Todo este proceso es transparente al programador.

Cuando en un método se hace referencia a un elemento, sea atributo o método, el compilador lo busca en el ámbito más cercano. Primero lo intenta localizar en el bloque de código que se está ejecutando, si no lo encuentra lo busca en método y, por último, en la clase a la que pertenece. Por esta razón no es necesario utilizar *this* para acceder a los miembros de la clase desde los métodos de la propia clase.

A primera vista, no parece muy útil. No obstante, en algunas circunstancias tendremos que utilizarlo.

Por ejemplo, si en el método *setDia* de la clase Fecha el parámetro de tipo int que recibe en lugar de llamarse *d* se hubiese llamado *día* en ese caso sí hubiese sido necesario utilizar *this*:

```
public void setDia(int dia) {
        this.dia = dia;
}
```

Como coincide el nombre del parámetro con el del atributo del objeto, si no utilizamos *this* el compilador entendería que hacemos el parámetro *dia* igual a sí mismo.

En este caso *this.dia* está indicando el atributo *día* del objeto que está usando el método *setDia*. Recuerda que *this* contiene la dirección de memoria del objeto que ha invocado al método.

Por ejemplo, creamos un objeto Fecha con el constructor por defecto:

```
Fecha f = new Fecha();
```

Asignamos ahora al atributo *día* del objeto *f* el valor 7:

```
f.setDia(7);
```

Al invocar al método f.setDia(7) *this* recibe la dirección de memoria del objeto f. Esto lo realiza java de forma interna. De este modo la instrucción *this.dia = dia* asigna al atributo *día* del objeto *f* el valor recibido, en este caso 7.

Si creamos otro objeto fecha distinto y le asignamos por ejemplo 20 como valor de día:

```
Fecha f1 = new Fecha();
f1.setDia(20):
```

En este caso al invocar al método f1.setDia(20) *this* recibe la dirección de memoria del objeto f1. De este modo la instrucción *this.dia = dia* ahora asigna al atributo *día* del objeto *f1* el valor recibido, en este caso 20.

this no se puede aplicar a atributos o métodos static ya que son métodos y atributos de clase no de instancia.