POO y Clases - Introducción

1. Programación Orientada a Objetos (POO).

La POO parte de una idea básica: agrupar los datos y módulos que los manejan en una única entidad llamada **objeto**. En POO, todo gira en torno a los objetos: un programa es un objeto, que a su vez está formado de objetos; cada variable es un objeto; la pantalla, el teclado, los ficheros y, en general, todos los dispositivos son objetos y se tratan como tales.

2. Conceptos fundamentales.

Objetos: unidad que engloba dentro de sí un conjunto de datos y módulos necesarios para el tratamiento de esos datos. Cada objeto contiene datos y funciones.

Atributos: Son los datos incluidos en el objeto. Parecidas a las variables de C clásico pero incluidas en los objetos.

Métodos: Son módulos que pertenecen a los objetos y que manejarán los atributos.

Mensajes: El modo mediante el que se comunican los objetos. Consiste en llamar a un método de un objeto.

Interfaz: Las clases (y, por lo tanto, los objetos) tienen partes públicas y partes privadas. La parte pública es visible para todos los objetos, mientras que la parte privada es sólo visible para el propio objeto. A la parte pública de un objeto se le denomina interfaz.

3. Clases

Las clases son el mecanismo por el que se pueden crear nuevos Tipos en Java. Las clases son el punto central sobre el que giran la mayoría de los conceptos de la Orientación a Objetos.

Una clase es una agrupación de datos y de código que actúa sobre esos datos, a la que se le da un nombre.

Una clase contiene:

- Datos (se denominan Datos Miembro). Estos pueden ser de tipos primitivos o referencias.
- Métodos (se denominan Métodos Miembro).

La sintaxis general para la declaración de una clase es:

```
modificadores class nombre_clase {
        declaraciones_de_miembros ;
}
```

Por ejemplo:

```
class Punto {
     int x;
     int y;
}
```

Los modificadores son palabras clave que afectan al comportamiento de la clase. Los iremos viendo progresivamente en los sucesivos capítulos.

4. Objetos, miembros y referencias

Un objeto es una instancia (ejemplar) de una clase. La clase es la definición general y el objeto es la materialización concreta (en la memoria del ordenador) de una clase.

El fenómeno de crear objetos de una clase se llama instanciación.

Los objetos se manipulan con referencias. Una referencia es una variable que apunta a un objeto. Las referencias se declaran igual que las variables de Tipos primitivos (tipo nombre). Los objetos se crean (se instancian) con el operador de instanciación **new**.

Ejemplo:

```
Punto p;
p = new Punto();
```

La primera línea del ejemplo declara una referencia (p) que es de Tipo Punto. La referencia no apunta a ningún sitio. En la segunda línea se crea un objeto de Tipo Punto y se hace que la referencia p apunte a él. Se puede hacer ambas operaciones en la misma expresión:

```
Punto p = new Punto();
```

A los miembros de un objeto se accede a través de su referencia. La sintaxis es:

nombre_referencia.miembro

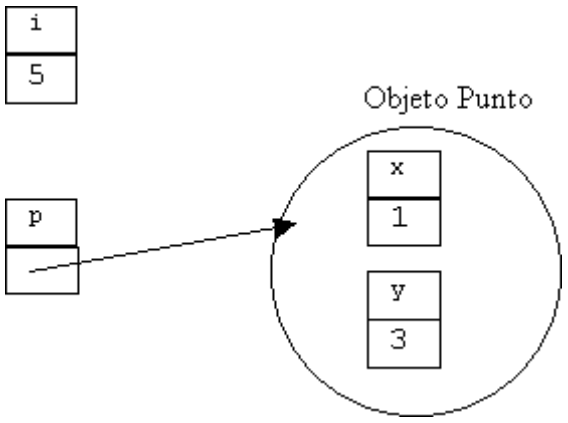
En el ejemplo, se puede poner:

```
p.x = 1;
p.y = 3;
```

Se puede visualizar gráficamente los datos primitivos, referencias y objetos de la siguiente forma:

- Datos primitivos: `int i = 5;`
- Referencias y objetos:

```
Punto p = new Punto();
p.x = 1;
p.y = 3;
```



Es importante señalar que en el ejemplo, p no es el objeto. Es una referencia que apunta al objeto.

Los métodos miembro se declaran dentro de la declaración de la clase, tal como se ha visto en el capítulo anterior. Por ejemplo:

```
class Circulo {
        Punto centro;   // dato miembro. Referencia a un objeto punto
        int radio;      // dato miembro. Valor primitivo
        float superficie() {                    // método miembro.
                return 3.14 * radio * radio;
        }                                       // fin del método
superficie
}                                               // fin de la clase
Circulo
```

El acceso a métodos miembros es igual que el que ya se ha visto para datos miembro. En el ejemplo:

```
Circulo c = new Circulo();
c.centro.x = 2;
c.centro.y = 3;
c.radio = 5;
float s = c.superficie();
```

Es interesante observar en el ejemplo:

- Los datos miembro pueden ser tanto primitivos como referencias. La clase Círculo contiene un dato miembro de tipo Punto (que es el centro del círculo).
- El acceso a los datos miembros del Punto centro se hace encadenando el operador. en la expresión `c.centro.x` que se podría leer como 'el miembro x del objeto (Punto) centro del objeto (Circulo) c'.
- Aunque el método superficie no recibe ningún argumento los paréntesis son obligatorios (Distinguen los datos de los métodos).
- Existe un Objeto Punto para cada instancia de la clase Circulo (que se crea cuando se crea el objeto Circulo).

5. Conceptos básicos. Resumen

- Una Clase es una definición de un nuevo Tipo, al que se da un nombre.
- Una Clase contiene Datos Miembro y Métodos Miembro que configuran el estado y las operaciones que puede realizar.
- Un Objeto es la materialización (instanciación) de una clase. Puede haber tantos Objetos de una Clase como resulte necesario.

- Los Objetos se crean (se les asigna memoria) con el Operador **new**.
- Los Objetos se manipulan con Referencias.
- Una Referencia es una Variable que apunta a un Objeto.
- El acceso a los elementos de un Objeto (Datos o métodos) se hace con el operador. (punto) : *nombre_referencia.miembro*

Clases - Constructores

---

1. Noción de constructor

Cuando se crea un objeto (se instancia una clase) es posible definir un proceso de inicialización que prepare el objeto para ser usado. Esta inicialización se lleva a cabo invocando un método especial denominado constructor. Esta invocación es implícita y se realiza automáticamente cuando se utiliza el operador **new**. Los constructores tienen algunas características especiales:

- El nombre del constructor tiene que ser igual al de la clase.
- Puede recibir cualquier número de argumentos de cualquier tipo, como cualquier otro método.
- No devuelve ningún valor (en su declaración no se declara ni siquiera **void**).

El constructor no es un miembro más de una clase. Sólo es invocado cuando se crea el objeto (con el operador **new**). No puede invocarse explícitamente en ningún otro momento.

Continuando con los ejemplos del capítulo anterior se podría escribir un constructor para la clase Punto, de la siguiente forma:

```
class Punto {
    int x , y ;
    Punto ( int a , int b ) {
        x = a ; y = b ;
    }
}
```

Con este constructor se crearía un objeto de la clase Punto de la siguiente forma:

```
Punto p = new Punto ( 1 , 2 );
```

2. Constructor no-args.

Si una clase no declara ningún constructor, Java incorpora un constructor por defecto (denominado constructor no-args) que no recibe ningún argumento y no hace nada.

Si se declara algún constructor, entonces ya no se puede usar el constructor no-args. Es necesario usar el constructor declarado en la clase.

En el ejemplo el constructor no-args sería:

```
class Punto {
    int x , y ;
```

```
    Punto ( ) { }
}
```

3. Sobrecarga de constructores.

Una clase puede definir varios constructores (un objeto puede inicializarse de varias formas). Para cada instanciación se usa el que coincide en número y tipo de argumentos. Si no hay ninguno coincidente se produce un error en tiempo de compilación.

Por ejemplo:

```
class Punto {
    int x , y ;
    Punto ( int a , int b ) {
        x = a ; y = b ;
    }
    Punto () {
        x = 0 ; y = 0;
    }
}
```

En el ejemplo se definen dos constructores. El citado en el ejemplo anterior y un segundo que no recibe argumentos e inicializa las variables miembro a 0. (Nota: veremos más adelante que este tipo de inicialización es innecesario, pero para nuestro ejemplo sirve).

Desde un constructor puede invocarse explícitamente a otro constructor utilizando la palabra reservada **this** . **this** es una referencia al propio objeto. Cuando **this** es seguido por paréntesis se entiende que se está invocando al constructor del objeto en cuestión. Puedes ver el uso más habitual de this aquí. El ejemplo anterior puede reescribirse de la siguiente forma:

```
class Punto {
    int x , y ;
    Punto ( int a , int b ) {
        x = a ; y = b ;
    }
    Punto () {
        this (0,0);
    }
}
```

Cuando se declaran varios constructores para una misma clase estos deben distinguirse en la lista de argumentos, bien en el número, bien en el tipo.

Esta característica de definir métodos con el mismo nombre se denomina sobrecarga y es aplicable a cualquier método miembro de una clase como veremos más adelante.

Clases - Miembros de Clase

---

1 Atributos de Clase

Un dato estático es una variable miembro que no se asocia a un objeto (instancia) de una clase, sino que se asocia a la clase misma; no hay una copia del dato para cada objeto sino una sola copia que es compartida por todos los objetos de la clase.

Por ejemplo:

```
class Punto {
    int x , y ;
    static int numPuntos = 0;

    Punto ( int a , int b ) {
        x = a ; y = b;
        numPuntos ++ ;
    }
}
```

En el ejemplo `numPuntos` es un contador que se incrementa cada vez que se crea una instancia de la clase Punto.

Obsérvese la forma en que se declara `numPuntos`, colocando el modificador **`static`** delante del tipo. La sintaxis general para declarar una variable es:

[*modificadores*] *tipo_variable nombre****;***

`static` es un modificador. En los siguientes capítulos se irán viendo otros modificadores. Los [ ] en la expresión anterior quieren decir que los modificadores son opcionales.

El acceso a las variables estáticas desde fuera de la clase donde se definen se realiza a través del nombre de la clase y no del nombre del objeto como sucede con las variables miembro normales (no estática). En el ejemplo anterior puede escribirse:

```
int x = Punto.numPuntos;
```

No obstante también es posible acceder a las variables estáticas a través de una referencia a un objeto de la clase. Por ejemplo:

```
Punto p = new Punto();
int x = p.numPuntos;
```

Las variables estáticas de una clase existen, se inicializan y pueden usarse antes de que se cree ningún objeto de la clase.

2 Métodos de Clase

Para los métodos, la idea es la misma que para los datos: los métodos de clase se asocian a una clase, no a una instancia.

Por ejemplo:

```
class Punto {
    int x , y ;
    static int numPuntos = 0;

    Punto ( int a , int b ) {
        x = a ; y = b;
        numPuntos ++ ;
    }

    static int cuantosPuntos() {
        return numPuntos;
    }
}
```

La sintaxis general para la definición de los métodos es, por tanto, la siguiente:

[modificadores] *Tipo_Valor_devuelto nombre_método* **(** *lista_argumentos* **) {**
    bloque_de_codigo;
}

El acceso a los métodos estáticos se hace igual que a los datos estáticos, es decir, usando el nombre de la clase, en lugar de usar una referencia:

```
int totalPuntos = Punto.cuantosPuntos();
```

Dado que los métodos estáticos tienen sentido a nivel de clase y no a nivel de objeto (instancia) los métodos estáticos no pueden acceder a datos miembros que no sean estáticos.

3 El método `main`

Un programa Java se inicia proporcionando al intérprete Java un nombre de clase. La JVM carga en memoria la clase indicada e inicia su ejecución por un método estático que debe estar codificado en esa clase. El nombre de este método es `main` y debe declararse de la siguiente forma:

```
static void main ( String [] args)
```

- Es un método estático. Se aplica por tanto a la clase y no a una instancia en particular, lo que es conveniente puesto que en el momento de iniciar la ejecución todavía no se ha creado ninguna instancia de ninguna clase.
- Recibe un argumento de tipo `String [].` String es una clase que representa una cadena de caracteres (se verá más adelante),
- Los corchetes `[]` indican que se trata de un array que se verán en un capítulo posterior.

No es obligatorio que todas las clases declaren un método `main` . Sólo aquellos métodos que vayan a ser invocados directamente desde la línea de comandos de la JVM necesitan tenerlo. En la práctica la mayor parte de las clases no lo tienen.

4 Inicializadores estáticos

En ocasiones es necesario escribir código para inicializar los datos estáticos, quizá creando algún otro objeto de alguna clase o realizando algún tipo de control. El fragmento de código que realiza esta tarea se llama inicializador estático. Es un bloque de sentencias que no tiene nombre, ni recibe argumentos, ni devuelve valor. Simplemente se declara con el modificador **`static`** .

La forma de declarar el inicializador estático es:

`static` { *bloque_codigo* }

Por ejemplo:

```
class Punto {
    int x , y ;
    static int numPuntos;

    static {
        numPuntos = 0;
    }

    Punto ( int a , int b ) {
        x = a ; y = b;
        numPuntos ++ ;
    }

    static int cuantosPuntos() {
        return numPuntos;
    }
}
```

Nota: El ejemplo, una vez más, muestra sólo la sintaxis y forma de codificación. Es innecesario inicializar la variable tal como se verá más adelante. Además podría inicializarse directamente con: `static int numPuntos = 0;`

Clases - Otros aspectos

---

1. Inicialización de variables

Desde el punto de vista del lugar donde se declaran existen dos tipos de variables:

- Variables miembro: Se declaran en una clase, fuera de cualquier método.
- Variables locales: Se declaran y usan en un bloque de código dentro de un método.

Las variables miembro son inicializadas automáticamente, de la siguiente forma:

- Las numéricas a 0.
- Las booleanas a **`false`**.
- Las char al carácter nulo (hexadecimal 0).

- Las referencias a **`null`**.

Nota: **`null`**es un literal que indica referencia nula.

Las variables miembro pueden inicializarse con valores distintos de los anteriores en su declaración.

Las variables locales no se inicializan automáticamente. Se debe asignarles un valor antes de ser usadas. Si el compilador detecta una variable local que se usa antes de que se le asigne un valor produce un error. Por ejemplo:

```
int p;
int q = p;     // error
```

El compilador también produce un error si se intenta usar una variable local que podría no haberse inicializado, dependiendo del flujo de ejecución del programa. Por ejemplo:

```
int p;
if (. . . ) {
     p = 5 ;
}
int q = p;     // error
```

El compilador produce un error del tipo 'La variable podría no haber sido inicializada', independientemente de la condición del if.

2. Ámbito de las variables

El ámbito de una variable es el área del programa donde la variable existe y puede ser utilizada. Fuera de ese ámbito la variable, o bien no existe o no puede ser usada (que viene a ser lo mismo).

El ámbito de una variable miembro (que pertenece a un objeto) es el de la usabilidad de un objeto. Un objeto es utilizable desde el momento en que se crea y mientras existe una referencia que apunte a él. Cuando la última referencia que lo apunta sale de su ámbito el objeto queda 'perdido' y el espacio de memoria ocupado por el objeto puede ser recuperado por la JVM cuando lo considere oportuno. Esta recuperación de espacio en memoria se denomina 'recogida de basura' y es descrita un poco más adelante.

El ámbito de las variables locales es el bloque de código donde se declaran. Fuera de ese bloque la variable es desconocida.

Ejemplo:

```
{
    int x;      // empieza el ámbito de x. (x es conocida y
utilizable)
    {
        int q;     // empieza el ámbito de q. x sigue siendo conocida.
        . . .
    }             // finaliza el ámbito de q (termina el bloque de
código)
     . . .         // q ya no es utilizable
}                 // finaliza el ámbito de x
```

3. Recogida de basura

Cuando ya no se necesita un objeto simplemente puede dejar de referenciarse. No existe una operación explícita para 'destruir' un objeto o liberar el área de memoria usada por él.

La liberación de memoria la realiza el recolector de basura (*garbage collector*) que es una función de la JVM. El recolector revisa toda el área de memoria del programa y determina que objetos pueden ser borrados porque ya no tienen referencias activas que los apunten. El recolector de basura actúa cuando la JVM lo determina (tiene un mecanismo de actuación no trivial).

En ocasiones es necesario realizar alguna acción asociada a la acción de liberar la memoria asignada al objeto (como por ejemplo liberar otros recursos del sistema, como descriptores de ficheros). Esto puede hacerse codificando un método `finalize` que debe declararse como:

```
protected void finalize() throws Throwable { }
```

Nota: las clausulas **`protected`** y **`throws`** se explican en capítulos posteriores.

El método `finalize` es invocando por la JVM antes de liberar la memoria por el recolector de basura, o antes de terminar la JVM. No existe un momento concreto en que las áreas de memoria son liberadas, sino que lo determina en cada momento la JVM en función de sus necesidades de espacio.

4. Sobrecarga de métodos

Una misma clase puede tener varios métodos con el mismo nombre siempre que se diferencien en el tipo o número de los argumentos. Cuando esto sucede se dice que el método está sobrecargado. Por ejemplo, una misma clase podría tener los métodos:

```
int metodoSobrecargado() { . . .}
int metodoSobrecargado(int x) { . . .}
```

Sin embargo no se puede sobrecargar cambiando sólo el tipo del valor devuelto. Por ejemplo:

```
int metodoSobrecargado() { . . .}
void metodoSobrecargado() { . . .}  // error en compilación
```

con esta definición, en la expresión `y.metodoSobrecargado()` la JVM no sabría que método invocar.

Se puede sobrecargar cualquier método miembro de una clase, así como el constructor.

5. La referencia `this`

En ocasiones es conveniente disponer de una referencia que apunte al propio objeto que se está manipulando. Esto se consigue con la palabra reservada **`this`**. **`this`** es una referencia implícita que tienen todos los objetos y que apunta a sí mismo. Por ejemplo:

```
class Circulo {
    Punto centro;
    int radio;
    . . .
    Circulo elMayor(Circulo c) {
        if (radio > c.radio) return this;
        else return c;
```

```
    }
}
```

El método elMayor devuelve una referencia al círculo que tiene mayor radio, comparando los radios del Círculo c que se recibe como argumento y el propio. En caso de que el propio resulte mayor el método debe devolver una referencia a sí mismo. Esto se consigue con la expresión **`return this.`**

6. La referencia `null`

Para asignar a una referencia el valor nulo se utiliza la constante null. El ejemplo del caso anterior se podría completar con:

```
class Circulo {
    Punto centro;
    int radio;
    . . .
    Circulo elMayor(Circulo c) {
        if (radio > c.radio) return this;
        else if (c.radio > radio) return c;
              else return null;
    }
}
```

7. Ocultamiento de variables

Puede ocurrir que una variable local y una variable miembro reciban el mismo nombre (en muchos casos por error). Cuando se produce esto la variable miembro queda oculta por la variable local, durante el bloque de código en que la variable local existe y es accesible. Cuando se sale fuera del ámbito de la variable local, entonces la variable miembro queda accesible. Obsérvese esto en el ejemplo siguiente:

```
. . .
String x = "Variable miembro";
. . .
void variableOculta() {
    System.out.println(x);
    {
        String x = "Variable local";
        System.out.println(x);
    }
    System.out.println(x);
}
```

La llamada al método `variableOculta()` producirá la siguiente salida:

```
Variable miembro
Variable local
Variable miembro
```

Se puede acceder a la variable miembro oculta usando la referencia this. En el ejemplo anterior la expresión:

```
System.out.println(this.x);
```

siempre producirá la salida '`Variable miembro`', puesto que `this.x` se refiere siempre a la variable miembro.