1.2 Iniciación a la Programación Orientada a Objetos

1. Subrutinas

Una subrutina o subprograma, como idea general, se presenta como un subalgoritmo que forma parte del algoritmo principal, el cual permite resolver una tarea específica.

Podemos distinguir tres términos que poseen diferencias:

1. **Función**: conjunto de instrucciones que devuelven un resultado.
2. **Procedimiento**: conjunto de instrucciones que se ejecutan sin devolver ningún resultado.
3. **Método**: función o procedimiento que pertenece a un objeto.

2. Funciones

Una función se construye de la siguiente manera:

```
modificador_acceso tipo_resultado nombre_función (tipo_parámetro
nombre_parámetro, ... ) {
   instrucciones
   return expresión;
}
```

- **modificador_acceso**: es la visibilidad que posee la función (Lo veremos más adelante. De momento, lo utilizaremos como public).
- **tipo_resultado**: es el tipo del resultado que devuelve la función.

- **nombre_función**: es el nombre que identifica a la función. Utiliza la notación lowerCamelCase. Ejemplo: imprimirResultadoDecimal.
- **tipo_parámetro nombre_parámetro, ...**: puede ocurrir que la función necesite ciertos valores para efectuar la misión para la que ha sido creada. Por ejemplo, la función suma necesitaría los valores que tiene que sumar. En este caso, se deben indicar cada uno de dichos valores con sus tipos correspondientes. A estos valores se les conoce como parámetros de la función. Si la función no necesita parámetros, entonces solamente se ponen los paréntesis: `nombre_funcion( )`

A todo esto se le conoce como la **firma** (signature) de la función:

```
modificador_acceso tipo_resultado nombre_función (tipo_parámetro
nombre_parámetro, ... )
```

- **instrucciones**: instrucciones que conforman el algoritmo de la función, para que realice la misión para la que ha sido creada.
- **return expresión;** : el return es el que nos devuelve el resultado, por lo tanto la expresión que acompaña al return tiene que devolver un valor correspondiente al *tipo_resultado* indicado en la firma de la función.

Ejemplo de función con dos parámetros:

```
public int add(int sum1, int sum2) {
        return sum1 + sum2;
}
```

Puede ser también que haya más de un return. En ese caso, el flujo de ejecución abandona la función en cuanto ejecute el primer return.

En el caso de que estemos definiendo un **procedimiento**, no tendremos *return* ya que no devuelve ningún resultado y el tipo_resultado es *void*. Como por ejemplo `System.out.println`, que escribe en pantalla lo que recibe por parámetro pero no devuelve nada.

Una función o procedimiento nos permite que reutilicemos un algoritmo ya que se puede utilizar cuando nos haga falta. Para ello, solamente tendremos que **llamar a la función o al procedimiento** por su nombre y pasarle los parámetros en el mismo orden que se han definido y pertenecientes al mismo tipo de dato. En la llamada, dichos parámetros se llaman **argumentos**.

```
package tema1_2_IniciacionPOO;

public class Functions {

    public void show() {

        boolean isAnEvenNumber;
        int result;

        isAnEvenNumber = isEven(5);//Se llama a la función isEven con un valor de
5 en el argumento
        showBoolean(isAnEvenNumber);//Se llama al procedimiento showBoolean con
la variable isAnEvenNumber como argumento
        isAnEvenNumber = isEven(4);//Se llama a la función isEven con un valor de
4 en el argumento
```

```
        showBoolean(isAnEvenNumber);//Se llama al procedimiento showBoolean con la variable isAnEvenNumber como argumento

        result = add(5, 2);//Se llama a la función add con los valores 5 y 2 en los argumentos
        showInt(result);//Se llama al procedimiento showInt con la variable result como argumento

    }

    public boolean isEven(int n) {//Función que devuelve true si n es par

        boolean result;
        result = n % 2 == 0;
        return result;

    }

    public int add(int sum1, int sum2) {//Función que suma sum1 y sum2

        return sum1 + sum2;

    }

    public void showBoolean(boolean b) {//Procedimiento que muestra por consola el valor de b

        System.out.println(b);

    }

    public void showInt(int n) {//Procedimiento que muestra por consola el valor de n

        System.out.println(n);

    }

    public static void main(String[] args) {

        new Functions().show();

    }


}
```

3. Introducción a la Programación Orientada a Objetos

La Programación Orientada a Objetos (***POO***) es una técnica de programar aplicaciones basada en una serie de objetos independientes que se comunican entre sí.

A Java se le considera un lenguaje orientado a objetos ya que siempre que se crea un programa en Java, por simple que sea, se necesita declarar una clase, y el concepto de clase pertenece a la programación orientada a objetos.

Un ***objeto*** es un elemento del programa que integra sus propios datos y su propio funcionamiento. Es decir, un objeto está formado por datos (atributos o propiedades) y por las funciones que es capaz de realizar el objeto (métodos). Esta forma de programar se asemeja más al pensamiento humano. La cuestión es detectar adecuadamente los objetos necesarios para una aplicación. De hecho hay que detectar las distintas clases de objetos.

Una ***clase*** es lo que define a un tipo de objeto. Al definir una clase lo que se hace es indicar como funciona un determinado tipo de objeto. Luego, a partir de la clase, podremos crear objetos de esa clase, es decir, la clase es como un molde a partir del cual se crean los objetos que pertenecen a ella. Realmente la programación orientada a objetos es una programación orientada a clases. Es decir, lo que necesitamos programar es como funcionan las clases de objetos.

Por ejemplo, una clase podría ser la clase *Coche*. Cuando se defina esta clase, indicaremos los atributos o propiedades (como el color, modelo, marca, velocidad máxima,...) y los métodos (arrancar, parar, repostar, acelerar, frenar...). Todos los coches, es decir, todos los objetos de la clase *Coche*, tendrán esas propiedades y esos métodos. Para explicar la diferencia entre clase y objeto:

- la clase *Coche* representa a todos los coches.
- un coche concreto es un objeto, es decir, un ejemplar de una clase es un objeto. También se le llama a los objetos *instancias* de la clase. Este término procede del inglés, *instance*, que realmente significa ejemplar.

Por ejemplo, si quisiéramos crear el juego del parchís en Java, una clase sería la casilla, otra las fichas, otra el dado, etc. En el caso de la casilla, se definiría la clase para indicar su funcionamiento y sus propiedades, y luego se crearían tantos objetos casilla como casillas tenga el juego. Lo mismo ocurriría con las fichas, la clase ficha definiría las propiedades de la ficha (color y posición por ejemplo) y su funcionamiento mediante sus métodos (por ejemplo un método sería mover, otro llegar a la meta, etc), luego se crearían tantos objetos ficha como fichas tenga el juego.

4. Creación de objetos

Una vez definida la clase, ya se pueden crear objetos de la misma. Para crear un objeto, hay que declarar una variable cuyo tipo será la propia clase.

Si por ejemplo definiéramos una clase llamada *Vehicle* para modelar vehículos, para crear un objeto tendríamos que declarar una variable de tipo *Vehicle*:

```
Vehicle car; //car es una variable de tipo Vehicle
```

Una vez definida la variable, se le crea el objeto llamando a un método que se llama constructor. Un constructor es un método que se invoca cuando se crea un objeto y que sirve para inicializar los atributos del objeto y para realizar las acciones pertinentes que requiera el mismo para ser creado. El constructor tiene el mismo nombre que la clase y para invocarlo se utiliza el operador **new**.

```
car = new Vehicle(); //Vehicle() es un método constructor
```

También se puede hacer todo en la misma línea:

```
Vehicle car = new Vehicle();
```

5. Acceso a los atributos y métodos del objeto

Una vez creado el objeto, se puede acceder a sus **atributos** de la siguiente manera: `objeto.atributo`.

```
car.wheelCount = 4;//Se le asigna 4 al atributo número de ruedas de la variable car
```

Los **métodos** se utilizan de la misma forma que los atributos, a excepción de que los métodos poseen siempre paréntesis ya que son funciones que pertenecen a un objeto: `objeto.método(argumentos)`.

```
car.accelerate(30);
/*El coche incrementa su velocidad en 30. Es decir, si iba a 90km/h, después de ejecutar el método el coche va a 120km/h */
```

6. Sobrecarga de métodos

Java admite sobrecargar los métodos, es decir, crear distintas variantes del mismo método con el mismo nombre pero que se diferencien en el orden, tipo o número de los parámetros.

Por ejemplo, tenemos el método para sumar `add(int x,double y)`:

- Sí podríamos definir el método `add(double x,int y)` porque varía el orden de los parámetros.

Otro ejemplo donde tenemos el método `add(int x,int y)`:

- No podríamos definir otro método `add(int a,int b)` porque no varía el tipo ni el número de parámetros.
- Sí podríamos definir `add(int a)` y `add(int a,int b,int c)` porque el número de parámetros varía.
- También podríamos definir `add(int x,double y)` porque aunque no varíe el número de parámetros, sí varía uno de los tipos.

7. Métodos estáticos y dinámicos

A los métodos asociados a los objetos se les conoce como métodos dinámicos. Pero puede ocurrir que tengamos métodos que no estén asociados a ningún objeto, por ejemplo, métodos de utilidad general. Dichos métodos se les conoce como métodos estáticos y se definen con la palabra **static**. Al no estar asociados a ningún objeto, se utilizan con el nombre de la clase: `Clase.metodoEstático(argumentos)`. Ejemplos de llamadas a métodos dinámicos y estáticos:

- Llamada a método dinámico: `car.accelerate(30);` Esto es: `objeto.nombreDelMetodo();`
- Llamada a método estático: `Math.pow(2,3);` Es decir, `NombreDeLaClase.nombreDelMetodo();`

8. Api de Java

La **API de Java** es una interfaz de programación de aplicaciones (**API**, por sus siglas del inglés: Application Programming Interface) provista por los creadores del lenguaje de programación **Java**, que da a los programadores los medios para desarrollar aplicaciones **Java**.

Al instalar Java (el paquete JDK) en nuestro ordenador, además del compilador y la máquina virtual de Java se instalan bastantes más elementos. Entre ellos, una cantidad muy importante de clases que ofrece la multinacional desarrolladora de Java y que están a disposición de todos los programadores listas para ser usadas. Estas clases junto a otros elementos forman lo que se denomina API de Java.

Los paquetes donde se encuentran dichas clases los podemos encontrar en https://docs.oracle.com → *Java* → *Java SE Documentation* → *Java SE Technical Documentation* → *JDK* de la versión deseada → *Specifications* → *API Documentation* → *Module: java.base*.

Otra manera de acceder rápido es poniendo en un buscador de Internet `Api Java Version Clase`, como por ejemplo: *Api Java 12 Math*.

9. La clase Math

La clase Math contiene los métodos para realizar operaciones matemáticas básicas, como potencias, logaritmos, raices cuadradas y funciones trigonométricas.

Si observamos esta clase en la API, todos los métodos tienen al principio la palabra *static,* ya que son métodos estáticos porque son funciones de utilidad que no se utilizan asociadas a un objeto. Después de la palabra static nos encontramos con el tipo del resultado que devuelve el método. Y a continuación, nos encontramos con el nombre del método y sus parámetros. Ejemplo: `static double abs(double a)`

```
package tema1_2_IniciacionPOO;

public class MathClass {

    public void show() {

        System.out.println(Math.abs(-3.2));
        System.out.println(Math.pow(2, 3));
        System.out.println(Math.sqrt(16));
        System.out.println(Math.min(20, 5));

    }

    public static void main(String[] args) {

        new MathClass().show();

    }

}
```

Obsérvese en la API la sobrecarga de los métodos *abs* y *min*.

10. La clase String

El texto es uno de los tipos de datos más importantes y por ello Java lo trata de manera especial. Para Java, las cadenas de texto son objetos especiales. Los textos deben manejarse creando objetos de tipo **String**.

Las cadenas se pueden inicializar de dos maneras:

- Usando el operador asignación: `String s="hola";`
- Usando el constructor: `String s=new String("hola");`

Los **literales** cadena se escriben entre comillas dobles: `"Esto es un literal cadena"`.

En java existe también la **cadena vacía**, es decir, una cadena sin ningún carácter.

Ejemplo: `String s="";` A la variable *s* se le está asignando la cadena vacía.

Como vimos en el Tema 1.1 Introducción al lenguaje Java, el operador concatenación `+` es un operador binario que devuelve una cadena resultado de concatenar las dos cadenas que actúan como operandos. Si sólo uno de los operandos es de tipo cadena, el otro operando se convierte implícitamente en tipo cadena.

Obsérvese en la API el método **valueOf**: es estático y está sobrecargado. Sirve para obtener la representación String de un valor u objeto.

```
package tema1_2_IniciacionPOO;

public class StringClass1 {

    public void show() {

        int i = 100;
        String string1, string2, string3, string4;

        string1 = "Esto es un literal cadena"; //Se le da un valor inicial a la
cadena con el operador de asignación =
        System.out.println(string1);
        System.out.println(string1 + " al cual le hemos concatenado este literal
cadena"); //Se concatena otra cadena con el operador +

        string2 = "hola";
        string3 = " que tal";
        string4 = string2 + string3;
        System.out.println(string4);

        System.out.println(i + 100); //Suma de enteros
        System.out.println(String.valueOf(i) + 100); //Concatenación de cadenas

    }

    public static void main(String[] args) {

        new StringClass1().show();

    }
```

```
}
```

Otros métodos de las cadenas muy útiles son:

- **charAt**: devuelve el carácter de la cadena del especificado índice. Dicho índice empieza en cero, es decir, con el cero se obtiene el primer carácter de la cadena.
- **length**: devuelve la longitud de la cadena.
- **equals**: compara si dos cadenas son iguales.

```
package tema1_2_IniciacionPOO;

public class StringClass2 {

    public void show() {

        String string = "hola";
        System.out.println(string.charAt(0));//h
        System.out.println(string.charAt(1));//o
        System.out.println(string.charAt(2));//l
        System.out.println(string.charAt(3));//a
        System.out.println(string.length());//4
        System.out.println(string.equals("hola"));//true
        System.out.println(string.equals("adiós"));//false

        //También se le pueden aplicar métodos a un literal cadena:
        System.out.println("hola".equals("hola")); //true
        System.out.println("adios".equals("hola"));//false

    }

    public static void main(String[] args) {

        new StringClass2().show();

    }

}
```

11. Wrappers

En ocasiones es muy conveniente poder tratar los datos primitivos (int, boolean, etc.) como objetos. Pero los datos primitivos no son objetos. Para resolver esta situación, la API de Java incorpora las clases envoltorio (**wrapper class**), que no son más que dotar a los datos primitivos con un envoltorio que permita tratarlos como objetos. Las clases envoltorio proporcionan métodos de utilidad para la manipulación de datos primitivos (conversiones de / hacia datos primitivos, conversiones a String, etc).

La siguiente tabla muestra los tipos primitivos y sus wrappers asociados:

Tipo primitivo	Wrapper asociado
byte	Byte

Tipo primitivo	Wrapper asociado
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

La clase `Number` es padre de todas las clases de números, es decir, de todas las clases de la tabla anterior excepto `Character` y `Boolean`.

Como ocurre con las cadenas, los wrappers también se pueden inicializar con el operador asignación:

```
Integer integer1 = 5;
```

Todos estos wrappers los encontraremos en la API de Java. Por ejemplo, si observamos en la API la clase Integer, podemos ver la siguiente firma de método: `static int parseInt(String s)` que convierte la cadena pasada por parámetro a entero.

```
package tema1_2_IniciacionPOO;

public class IntegerClass {

    public void show() {

        Integer integer1, integer2;
        int i;
        integer1 = 5;
        System.out.println(integer1);
        i = Integer.parseInt("7");//Convierte la cadena a int. Método estático
por lo que se utiliza con Integer
        System.out.println(i);
        integer2 = Integer.valueOf(i);//Convierte el int a Integer. También es
estático
        System.out.println(integer2);
        i = integer1.intValue();//Convierte el Integer a int. Método dinámico por
lo que se utiliza con el objeto
        System.out.println(i);

    }

    public static void main(String[] args) {

        new IntegerClass().show();

    }
```

```
}
```

En Java, los conceptos de **autoboxing** y **unboxing** se refieren a la conversión automática entre los tipos de datos primitivos (como `int`, `double`, etc.) y sus correspondientes clases envolventes (como `Integer`, `Double`, etc.). Estas clases envolventes, o *wrappers*, permiten que los tipos primitivos sean tratados como objetos, lo que es necesario en ciertas situaciones, como cuando se trabaja con colecciones de clases genéricas (`ArrayList`, `HashMap`, etc.), que no pueden contener tipos primitivos directamente.

11.1 Autoboxing

Autoboxing es el proceso automático mediante el cual Java convierte un tipo primitivo en su correspondiente tipo de clase envolvente. Esto ocurre, por ejemplo, cuando intentas asignar un valor primitivo a una variable que espera un objeto de la clase envolvente.

Ejemplo:

```
int num = 10;
Integer obj = num;  // Aquí se realiza autoboxing de int a Integer
```

11.2 Unboxing

Unboxing es el proceso inverso, donde un objeto de una clase envolvente se convierte automáticamente en su tipo primitivo correspondiente.

```
Integer obj = 20;
int num = obj;  // Aquí se realiza unboxing de Integer a int
```

En este caso, el objeto `obj` de tipo `Integer` se convierte automáticamente en un valor primitivo de tipo `int` mediante el proceso de unboxing.

11.3 El método estático valueOf

En versiones anteriores de Java, los objetos de las clases envolventes (wrappers) como *Integer*, *Double*, *Boolean*, etc., se podían crear utilizando constructores estándar. Sin embargo, a partir de Java 5, se introdujo el método de fábrica estático `valueOf` como una alternativa preferida para crear instancias de estos tipos envolventes.

Los métodos `valueOf` implementan un patrón conocido como *interning*, que permite reutilizar instancias previamente creadas en lugar de crear un nuevo objeto cada vez. Esto mejora la eficiencia, especialmente cuando se trabajan con valores pequeños o comunes, reduciendo la sobrecarga de creación de objetos y el consumo de memoria. Esto es particularmente útil para tipos como *Boolean*, donde solo existen dos posibles valores (*true* y *false*).
`Boolean.valueOf(true)` siempre devuelve la misma instancia de `Boolean.TRUE`, mientras que un constructor `new Boolean(true)` crearía un nuevo objeto cada vez.

Ejemplo:

```
Boolean x = Boolean.valueOf(true);
Boolean y = Boolean.valueOf(true);
System.out.println(x == y);  // True, ambos referencian el mismo objeto
```

Veamos un ejemplo comparativo entre el uso de un constructor y el método `valueOf`:

Uso del Constructor:

```
Integer x = new Integer(10);
Integer y = new Integer(10);
System.out.println(x == y);  // False, se crean dos objetos distintos
```

Uso de `valueOf`:

```
Integer x = Integer.valueOf(10);
Integer y = Integer.valueOf(10);
System.out.println(x == y);  // True, se reutiliza el mismo objeto
```

La sustitución de los constructores por métodos de fábrica estáticos como `valueOf` es parte de una tendencia en Java hacia la optimización de recursos y la eficiencia en la gestión de memoria. Estos métodos proporcionan beneficios significativos en términos de rendimiento y claridad, y son una parte fundamental del manejo moderno de objetos en Java, especialmente en lo que respecta a los tipos envolventes. Aunque los constructores aún están disponibles, su uso no es recomendable en la mayoría de los casos debido a las ventajas que ofrece `valueOf`.

12. Encadenamiento de llamadas a métodos

Se emplea cuando invocamos a un método de un objeto que nos devuelve como resultado otro objeto al que podemos volver a invocar otro método y así encadenar varias operaciones.

```
package tema1_2_IniciacionPOO;

public class CallsToMethods {

    public void showCallsToMethods() {

        Boolean b;
        String string;

        string = "EntornosDeDesarrollo";
        System.out.println(string.substring(10).toUpperCase()); //DESARROLLO

        b = Boolean.TRUE;
        System.out.println(b.toString().charAt(2)); //u

    }

    public static void main(String[] args) {

        new CallsToMethods().showCallsToMethods();

    }

}
```

En este ejemplo, el método `substring(10)` está devolviendo una subcadena de la cadena string a partir del carácter 10 empezando en 0, es decir, "Desarrollo", al que se le invoca luego el método `toUpperCase` devolviendo como resultado la cadena "DESARROLLO".

Y el método `toString()` de la variable b de tipo Boolean está devolviendo la cadena "true" a la que se le encadena el método `charAt(2)` devolviendo el carácter 'u'.

13. Números aleatorios

En java disponemos de la clase **Random** para generar números aleatorios. La clase dispone de dos constructores, uno sin parámetros y otro con un parámetro llamado semilla (**seed**). Aunque no podemos predecir que números se generarán con una semilla particular, podemos sin embargo, duplicar una serie de números aleatorios usando la misma semilla. Es decir, cada vez que creamos un objeto de la clase Random con la misma semilla obtendremos la misma secuencia de números aleatorios. Podemos cambiar la semilla de los números aleatorios en cualquier momento utilizando el método *setSeed*.

Algunos métodos de la clase Random para generar números aleatorios:

- nextInt(): genera un número aleatorio entero de tipo int
- nextInt(int n): genera un número aleatorio entero de tipo int entre 0(incluido) y n(excluido)
- nextLong(): genera un número aleatorio entero de tipo long
- nextFloat(): genera un número aleatorio de tipo float entre 0.0(incluido) y 1.0(excluido)
- nextDouble(): genera un número aleatorio de tipo double entre 0.0(incluido) y 1.0(excluido)
- nextBoolean(): genera un booleano aleatorio

Para generar números aleatorios enteros comprendidos entre un rango de dos números enteros *min* y *max* ambos incluidos, hay que utilizar la siguiente fórmula:

`nextInt(max - min + 1) + min`

Por ejemplo, si queremos generar un número aleatorio entre 5 y 10: nextInt(10-5+1)+5 → nextInt(6)+5

En el siguiente ejemplo, se puede observar en la ejecución que utilizando el Random con semilla siempre se generan los mismos números:

```
package tema1_2_IniciacionPOO;

import java.util.Random;

public class RandomClass {

    public void show() {

        Random random = new Random(); // Sin semilla
        Random randomSeed = new Random(3816);// Con semilla. Siempre se generan
los mismos números

        System.out.println(random.nextBoolean());
        System.out.println(random.nextInt());
        System.out.println(random.nextLong());
        System.out.println(random.nextFloat());
        System.out.println(random.nextDouble());
```

```
        System.out.println(random.nextInt(6) + 5);// Genera un número aleatorio
entre 5 y 10

        System.out.println(randomSeed.nextBoolean());
        System.out.println(randomSeed.nextInt());
        System.out.println(randomSeed.nextLong());
        System.out.println(randomSeed.nextFloat());
        System.out.println(randomSeed.nextDouble());
        System.out.println(randomSeed.nextInt(6) + 5);

    }

    public static void main(String[] args) {

        new RandomClass().show();

    }

}
```