Unidad IV

Métodos.

4.1 Definición de un método.

El polimorfismo, en programación orientada a objetos, se refiere a la posibilidad de acceder a un variado rango de funciones distintas a través del mismo interfaz. O sea, un mismo identificador puede tener distintas formas (distintos cuerpos de función, distintos comportamientos) dependiendo del contexto en el que se halle. El polimorfismo se puede establecer mediante sobrecarga, sobreescritura y enlace dinámico.

Sobrecarga

Este término se refiere al uso del mismo identificador u operador en distintos contextos y con distintos significados. Si para cada funcionalidad necesitada fuese necesario escribir un método, el código resultante sería inmanejable. Supongamos que los desarrolladores de Java hubiesen creado un método para escribir en pantalla una cadena de texto, otro diferente para escribir un entero, otro para un doble, y así para todas las combinaciones posibles, sería casi imposible conocer dichos métodos en totalidad. En cambio, con “System.out.print()” o “System.out.println()” podemos escribir cualquier mensaje en pantalla.

Este tipo de codificación nos es permitido gracias a la sobrecarga, la cual se aplica a métodos y constructores.

La sobrecarga de métodos hace que un mismo nombre pueda representar distintos métodos con distinto tipo y número de parámetros, manejados dentro de la misma clase. En el ámbito de la POO, la sobrecarga de métodos se refiere a la posibilidad de tener dos o más métodos con el mismo nombre pero distinta funcionalidad. Es decir, dos o más métodos con el mismo nombre realizan acciones diferentes y el compilador usará una u otra dependiendo de los parámetros usados. Esto también se aplica a los constructores (de hecho, es la aplicación más habitual de la sobrecarga).

Sobreescritura

La sobreescritura se aplica a los métodos y está directamente relacionada a la herencia; se refiere a la redefinición de los métodos de la clase base en las subclases.

Enlace dinámico

Esto permite invocar operaciones en objetos obviando el tipo actual de éstos hasta el momento de ejecutar el código. O sea, nos permite definir elementos como un tipo e instanciarlos como un tipo heredado.

4.2 Estructura de un método.

- Casi todos los métodos tienen una lista de parámetros; los parámetros de un método también son variables locales.
- Es recomendable limitar los métodos a realizar una sola tarea bien definida y el nombre del método debe expresar efectivamente dicha tarea.
- El *nombre* es cualquier identificador válido.
- El *tipo-de-valor-devuelto* es el tipo de dato del resultado, que el método devuelve al invocador.
- El *tipo-de-valor-devuelto* void indica que el método no devuelve ningún valor.
- Omitir el *tipo-de-valor-devuelto* en una definición de método causa un error de sintaxis.
- Olvidar devolver un valor desde un método que se supone debe devolver un valor es un error de sintaxis.

4.3 Valor de retorno.

- La llamada o invocación a un método se puede realizar de dos formas, dependiendo de que el método devuelva o no un valor:

    1. Si el método devuelve un valor, la llamada al método se trata normalmente como un valor.

Ejemplo N.1 :

```
int mayor = max(3,4);

// Se llama al método max(3,4) y asigna el resultado del método a la variable mayor.
```

Ejemplo N.2:

```
System.out.println(max(3,4));

// Imprime el valor devuelto por la llamada al método max(3,4)
```

1. Si el método devuelve void, una llamada al método debe ser una sentencia.

Ejemplo N.1 :

```
System.out.println("Invocación");

// El método println ( ) devuelve void.
```

Ejemplo N.2:

```
depositar( );

// Invocación al método depositar.
```

4.4 Declaración de un método.

Cuando uno plantea una clase en lugar de especificar todo el algoritmo en un único método (lo que hicimos en los primeros pasos de este tutorial) es dividir todas las responsabilidades de las clase en un conjunto de métodos.

Un método hemos visto que tiene la siguiente sintaxis:

```
public void [nombre del método]() {
  [algoritmo]
}
```

Veremos que hay varios tipos de métodos:

Métodos con parámetros.

Un método puede tener parámetros:

```
public void [nombre del método]([parámetros]) {
  [algoritmo]
```

}

Los parámetros los podemos imaginar como variables locales al método, pero su valor se inicializa con datos que llegan cuando lo llamamos.

Problema 1:

Confeccionar una clase que permita ingresar valores enteros por teclado y nos muestre la tabla de multiplicar de dicho valor. Finalizar el programa al ingresar el -1.

4.5 Ámbito y tiempo de vida de variables.

- El ámbito de una variable u objeto es el espacio del programa en el que esa variable existe. Por ello, se habla de “ámbito de vida”
- De forma general (hay excepciones que veremos más adelante), la vida de una variable comienza con su declaración y termina en el bloque en el que fue declarada (los bloques de código se delimitan por llaves: {}). Por ejemplo, ¿cuál es el ámbito de la variable ‘radio’ y del vector ‘args’?:

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

{

   double PI = 3.1416;

   double radio = 3;

   System.out.println( “El área es” + (PI*radio*radio) );

}
```

- Más adelante profundizaremos en los diferentes tipos de ámbito
- Ya vimos que el ámbito de una variable u objeto es el espacio del programa en el que esa variable existe. Por ello, se habla de “ámbito de vida”
- Los principales tipos de ámbitos son:
    - Ámbito de objeto. Los atributos de un objeto (que no son static) viven en el espacio de vida del objeto y son accesibles por cualquier método del objeto (siempre que el método no sea static). Por ello, a veces se llaman variables de objeto o variables de instancia

- Ámbito de método. Variables y objetos declarados en un método. Su ámbito de vida se ciñe al método en el que fueron declaradas, por ello a veces se llaman variables de método o función
- Ámbito de clase. Las variables static viven con independencia de que hayamos hecho instancias de la clase. Podemos acceder a ellas (si son públicas) usando el nombre de la clase y viven desde que se declara la clase, por ello se llaman variables de clase

4.6 Argumentos y paso de parámetros.

Las variables en la lista de parámetros se separan con comas.

Los parametros de la lista en la especificación del método, son llamados parámetros formales.

Cuando un método es llamado, estos parámetros formales son reemplazados por los parámetros actuales.

Los parámetros actuales deben ser equivalentes en tipo, orden y número a los parametros formales.

Cuando es invocado un método con un parámetro de tipo primitivo, tal como “int”, el valor del parámetro actual es pasado al método.

El valor actual de la variable fuera del método no es afectado, independientemente de los cambios hechos al parámetro formal dentro del método.

4.7 Sobrecarga de métodos.

Tanto en Java como en C++ es posible definir varios métodos para la misma clase con el mismo nombre pero con diferencias en:

En la lista de argumentos:

- el número
- el tipo de los argumentos
- o ambos

Esto se conoce como *sobrecarga de métodos*.

Si se tratan de crear dos métodos con la misma interfaz y diferentes tipos de retorno, la clase no se compila.

- Los constructores, como cualquier otro método, pueden ser sobrecargados:

```
class Circulo {
        private double radio;
        private double PI = 3.1416;
        /****** Constructores sobrecargados ***/
        public Circulo() { }
        public Circulo( double nuevoRadio ) {
                setRadio( nuevoRadio );
        }
        public Circulo( Circulo circulo ) {
                setRadio( circulo.getRadio() );
        }
        ...
}
```

- Observe que el último constructor define el radio a partir del radio de otro círculo, que se ha pasado como argumento

4.8 Encapsulamiento.

Es una de las propiedades de la POO que define que el objeto debe ser una cápsula o caja negra que encapsula su funcionamiento y estructura interna.

Sólo se ven desde afuera los miembros (la interfaz para el caso de los métodos) con visibilidad pública.

Buenas Prácticas: los atributos deben ser privados o

protegidos y se debe acceder a ellos a través de los métodos que pueden ser públicos.

El encapsulamiento permite:

- Ocultar detalles de implementación.
- Simplificar el programa.
- Minimizar el impacto del cambio.

Garantizar integridad de los datos