Contenido

1. PROCEDIMIENTOS Y FUNCIONES

La programación estructurada define el concepto de módulo como una *secuencia de instrucciones físicamente contiguas y lógicamente vinculadas, que pueden ser invocadas mediante un nombre desde distintos puntos del programa.*

En los lenguajes estructurados, el concepto de módulo se materializa en forma de una construcción denominada, genéricamente, ***rutina*** o ***subprograma***, que es un bloque de instrucciones al que se asigna un identificador para poder ser invocado desde distintas partes del programa

En los lenguajes orientados a objetos, el concepto de rutina o subprograma se materializa en forma de **métodos**, que son las acciones que puede realizar un objeto. Los métodos pueden clasificarse en dos tipos: ***procedimientos*** y ***funciones***.

- Un **procedimiento** es un conjunto de instrucciones que tiene asignado un identificador y puede recibir, opcionalmente, un conjunto de parámetros que se utilizarán para realizar una determinada tarea.

- Una **función** es similar a un procedimiento: está formada por un conjunto de instrucciones que pueden ser invocadas mediante un identificador. Al igual que un procedimiento, una función puede recibir, opcionalmente, un conjunto de parámetros. Sin embargo, a diferencia de lo que ocurre con un procedimiento, una función **devuelve un valor** al código llamador que la ha invocado.

La sintaxis básica para la declaración de una función o procedimiento es la siguiente:

```
<tipo_de_dato_devuelto | void> <identificador>
([modificadores] [lista de parámetros] )
   {
   [sentencias]
   [return <valor de retorno>;]
   }
```

Si el método es un procedimiento, el tipo de dato devuelto debe ser `void`, que indica que no se devuelve información al código llamador. Por el contrario, si el método es una función, en vez de `void`, se debe indicar el tipo de dato que devuelve la función al código llamador.

A diferencia de lenguajes clásicos, como C, la ubicación dentro de la clase en la que se define un método es irrelevante. Como hemos visto en documentos previos, la visibilidad de un método se extiende a toda la clase, sin importar en qué punto de ésta se escribe. No es necesario, por tanto, escribir los métodos en un orden determinado.

A continuación se muestra un ejemplo de procedimiento y de función:

```
// Este procedimiento muestra un mensaje por consola. Como se trata de un
// procedimiento, no devuelve un valor al código llamador. Para indicar que no
// devuelve un valor al código llamador, se utiliza la palabra reservada 'void'
void showMessage()
   {
   System.out.print("Hi world!");
   }

// Esta función devuelve un dato de tipo String al código llamador. Observa que el
// tipo de dato devuelto al código llamador se especifica delante del
// identificador de la función:
string getMessage()
   {
   string message = "Hi world!";
   return message;
   }
```

2. SENTENCIA DE LLAMADA O *INVOCACIÓN*

Para ejecutar un método es necesario escribir una ***sentencia de llamada*** o ***invocación***. La invocación de un método permite transferir el camino de ejecución a ese método, pasando los parámetros (si existen) desde el código llamador al método.

La invocación se realiza escribiendo el nombre de la rutina seguida de la lista de parámetros, entre paréntesis, que espera recibir el método (si es que recibe parámetros). Los paréntesis deben especificarse en cualquier caso, incluso si no se pasan parámetros a la rutina.

La sintaxis para invocar a un método regular es la siguiente:

```
<id_objeto>.<id_método>([modificadores] [lista de parámetros]);
```

Si, por el contrario, queremos invocar a un método estático, hemos de tener en cuenta que no se puede acceder a éste a través de una referencia a un objeto, sino a través del identificador de la clase:

```
<id_clase>.<id_método>([modificadores] [lista de parámetros]);
```

La sentencia de llamada puede aparecer de forma repetida y en distintas partes del programa.

Los parámetros en la sentencia de invocación deben ser proporcionados en el mismo orden y deben ser del mismo tipo que los que se definen en el método. Abordaremos estas cuestiones al tratar los tipos de paso de parámetros.

En este fragmento se muestra un ejemplo de invocación a un procedimiento (en este caso, un procedimiento estático):

```
class Program
   {
   public static void main(String[] args)
      {
      float f1 = 5.0f;
      float f2 = 3.0f;

      // La sentencia de llamada a un método se especifica escribiendo el
      // identificador del método, seguido de la lista de parámetros, si los hay,
      // separados por coma y entre paréntesis.
      add(f1, f2);
      }

   // Este procedimiento realiza la suma de dos números que recibe como
   // parámetros.
   // En el cuerpo del método se efectúa el cálculo y se muestra el resultado por
   // consola.
   // Observa que el método NO devuelve un valor al código llamador.
   static void add(float n1, float n2)
      {
      float result = n1 + n2;
      System.out.print(result);
      }
   }
```

La invocación a una función se realiza de la misma forma que en el caso de un procedimiento, con la salvedad de que la función devuelve un dato al código llamador. Normalmente, en el código llamador, vamos a capturar en una variable el dato devuelto por la función, para utilizar dicho valor de acuerdo al propósito del programa:

```
class Program
   {
   public static void main(String[] args)
      {
      float f1 = 5.0f;
      float f2 = 3.0f;

      // Esta sentencia efectúa la llamada a la función y, una vez que se ha
      // ejecutado, recoge el resultado que devuelve la función en la variable
      // result
      float result = add(f1, f2);

      // [ ... operaciones con la variable result ... ]
      }

   // Esta función realiza la suma de dos números que recibe como
   // parámetros. En el cuerpo de la función se efectúa el cálculo y se devuelve
   // el resultado al código llamador, que lo captura en una variable
   static float add(float n1, float n2)
      {
      float result = n1 + n2;
      return result;
      }
   }
```

En la función `add`, se devuelve al código llamador el valor almacenado en variable `result`, que está definida en el cuerpo del método `add`. En el código llamador se captura ese valor en otra variable, también llamada `result`. Ambas variables son distintas y sus identificadores no tienen por qué coincidir necesariamente. Se ha utilizado el mismo identificador para facilitar la lectura y el seguimiento del dato devuelto por la función. De hecho, en el cuerpo de la función `add` bastaría con haber escrito:

```
return n1+n2;
```

sin necesidad de declarar una variable.

3. FIRMA DE UN MÉTODO

La firma de un procedimiento o función es la descripción del tipo de dato devuelto por el método, el identificador del mismo y la lista de parámetros que recibe. El prototipo de una rutina se corresponde, por tanto, con la línea (o parte de la línea) en la que se define el método

A modo de ejemplo, dada la siguiente función que hemos definido en un ejemplo anterior:

```
static float add(float n1, float n2)
   {
   float result = n1 + n2;
   return result;
   }
```

Al hablar de firma, nos estamos refiriendo a lo siguiente:

```
add(float, float)
```

Fíjate en que **la firma está formada únicamente por el identificador del método y la lista de tipos de dato de los parámetros.**

Conocer la firma es importante, porque ayuda a entender si el método es una función o procedimiento, qué tipo de dato devuelve, qué parámetros recibe y cómo debe ser invocado ese método. Además, conocer la firma es importante, puesto que el compilador permite crear métodos ***sobrecargados***, es decir métodos que tienen el mismo identificador pero que difieren en su firma por tener distinto número y/o tipo de parámetros.

- Por ejemplo, el compilador **permitiría la coexistencia** de estas funciones, ya que tienen distinto prototipo:

```
int testMethod(string param1)
int testMethod(string param1, int param2)
```

- Sin embargo, las siguientes funciones **no podrían coexistir** en la misma clase, puesto que, para el compilador, tienen el mismo prototipo y son, por tanto, la misma función definida dos veces:

```
int testMethod(string param1)
string testMethod(string param1)
```

  Como puedes ver, el tipo de dato devuelto no forma parte del prototipo, por lo que, aunque ambas funciones tienen un tipo de retorno diferente, el compilador ignora esa parte.

- Las siguientes funciones **tampoco podrían coexistir** en la misma clase, puesto que el nombre de los parámetros no forma parte del prototipo:

```
string testMethod(string p1)
string testMethod(string param1)
```

Entender la información que proporciona el prototipo es esencial para poder diseñar correctamente procedimientos y funciones. También resulta esencial para comprender la documentación técnica y poder utilizar correctamente rutinas, tanto propias como las ofrecidas por una biblioteca.

4. PARÁMETROS REALES Y FORMALES. *MAPEO*.

Los parámetros que aparecen en la firma de un método se denominan ***parámetros formales***. Estos parámetros actúan como variables receptoras de los datos que pasa el código llamador al invocar al método.

Los datos concretos con los que se llama a un método se denominan ***parámetros reales***. Estos datos son proporcionados por el código llamador en la sentencia de llamada y se almacenan (se "***mapean***") en los parámetros formales durante la ejecución del método.

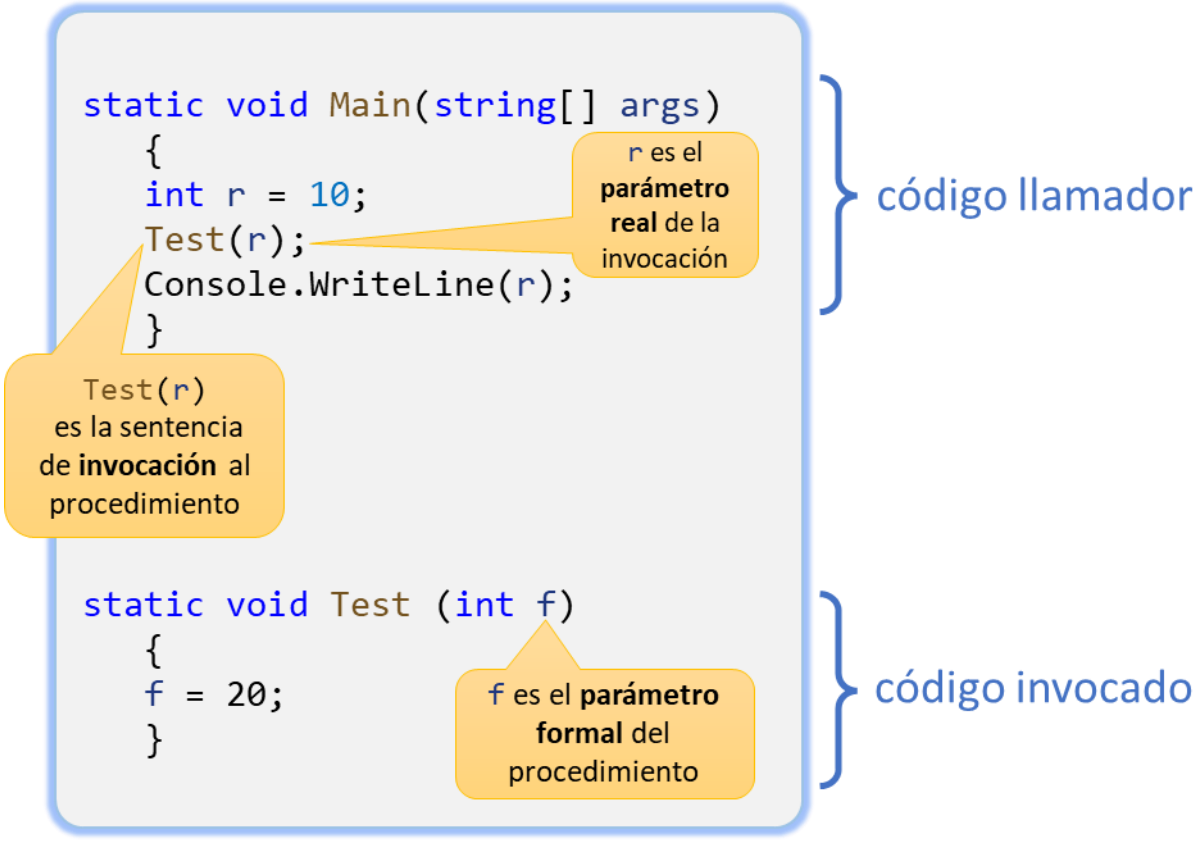
Por tanto, en cada invocación de un método, los parámetros formales almacenan los valores de los parámetros reales con los que ha sido invocada la rutina.

```
class Program
   {
   static void main(String[] args)
      {
      float f1 = 5.0f;
      float f2 = 3.0f;

      // f1 y f2 son los parámetros reales, es decir, los valores con los que
      // va a operar la función en esta invocación
      float result = add(f1, f2);
      }

      // n1 y n2 son los parámetros formales, que actúan como variables receptoras
      // de los parámetros reales que se pasan en la sentencia de invocación.
   static float add(float n1, float n2)
      {
      float resultado = n1 + n2;
      return resultado;
      }
   }
```

5. ESQUEMA DE LOS ELEMENTOS QUE INTERVIENEN EN LA INVOCACIÓN DE UN MÉTODO



6. POLIMORFISMO MEDIANTE MÉTODOS SOBRECARGADOS

El compilador de Java permite declarar múltiples ***sobrecargas*** (*overloads*) de un método. Un **método sobrecargado** tiene el mismo nombre que otro método de la clase, pero sus firmas difieren.

Generalmente, dos métodos sobrecargados realizan la misma función, pero recibiendo distintos parámetros de entrada, o bien realizan una función similar, pero con pequeñas variaciones en el comportamiento de los métodos.

A efectos de sobrecarga de métodos, el compilador **NO** considera que los siguientes elementos formen parte del prototipo del método:

- El tipo devuelto por el método.
- El modificador de accesibilidad.
- El modificador `final`.
- El modificador varargs (`...`).

Por tanto, para crear métodos sobrecargados es necesario que difieran en el **orden y/o número y/o tipo de parámetros**.

Los métodos sobrecargados constituyen una forma de ***polimorfismo***, como veremos en las siguientes unidades.