Capítulo 5. Métodos y programación estructurada

A. J. Pérez

Métodos y programación estructurada

Los subprogramas

El principio de modularidad - *divide y vencerás*- para la resolución de un problema es habitual en la vida cotidiana, especialmente si es un problema complejo. Según este principio **un problema se puede descomponer en varias partes más pequeñas y fáciles de resolver**. Las soluciones parciales adecuadamente coordinadas proporcionan la solución del problema completo.

Análogamente, cuando se escribe un programa como solución para un problema específico, se puede aplicar el mismo principio de modularidad para descomponer una tarea en *subtareas*, las soluciones de las subtareas se *coordinan* y *colaboran* en el programa completo. **Las soluciones de estas subtareas son los *subprogramas*.**

En programación estructurada clásica, los subprogramas pueden encontrarse bajos tres formas básicas:

- **Subrutinas**
  No utilizan argumentos y no devuelven un resultado.
- **Procedimientos**
  Reciben argumentos y no devuelven un resultado.
- **Funciones**
  Reciben argumentos y devuelven un resultado.

En Java los subprogramas son siempre parte de los objetos o las clases: además se denominan métodos, pudiendo actuar como cualquiera de las tres formas mencionadas.

Métodos

Un método es una parte integrante de un objeto que resuelve parte un problema.
Los métodos se encargan de realizar una parte del proceso realizado por el programa al que pertenecen; más adelante se verá que los métodos, desde la perspectiva de orientación a objetos, **representan además el comportamiento de una clase de objeto.**

En consecuencia, dado un conjunto de diferentes métodos, se pueden crear de forma sencilla programas relativamente grandes para resolver problemas complejos.

Ejemplo de método:

```
static double calcularAreaRectangulo(double base, double altura) {
	double area = base * altura;
	return area;
}
```

¿Por qué utilizar métodos?

Hay tres razones importantes que justifican la utilización de métodos:

1. **Legibilidad y estructuración aplicando los principio de Modularidad y de Delegación**
   Al crear un programa, es una buena práctica utilizar métodos para que sea más estructurado y fácil de leer, no sólo para el autor sino para otros programadores. El motivo de esto es que, estadísticamente, del total de tiempo que se dedica al desarrollo de un programa, sólo un 20% es escribir el código inicial; el resto se dedica a pruebas, depuración y a añadir nuevas características no incluidas en la versión inicial. En la mayoría de los casos, una vez que se escribe el código, la depuración y prueba no la realiza sólo el creador, sino también otros programadores. Por lo tanto, es importante que el programa esté redactado utilizando técnicas de código limpio que produzcan resultados bien estructurados y sea de fácil lectura.
2. **Evitar la duplicación del código**
   Cuando se utiliza un mismo código más de una vez en un programa, es muy recomendable definirlo como un método para poder utilizarlo en varios sitios sin repetir código; el programa es así más legible, mejor estructurado y fácil de mantener. Esto está directamente relacionado además con la reutilización.
3. **Reutilización de código aplicando el principio de Responsabilidad Única**
   La reutilización de código es uno de los grandes objetivos de todas las metodologías de programación; consiste en utilizar código elaborado para un programada en otro programa diferente. Si un programa está diseñado estructuradamente y es fácil de mantener, sus partes serán fácilmente aprovechables para otras aplicaciones diferentes.

En la utilización de métodos hay que distinguir tres aspectos:

- **Declaración**
  Consiste en especificar cómo se denomina un método y a qué forma de subprograma corresponde.
- **Implementación**
  Definir, crear o implementar un método consiste en la escritura real del código que resuelve la tarea específica prevista.
- **Llamada al método**
  Llamar o invocar a un método desencadena la ejecución del código escrito en la implementación del mismo. La llamada a un método siempre se realiza en la implementación de otro que delega alguna de sus partes de proceso.

Declaración

En Java, un método debe ser declarado dentro del contexto de una clase o estructura

equivalente y se debe seguir el principio de Mínimo Acoplamento.

Un ejemplo básico es el método `main()` que siempre se declarar e implementa dentro del bloque delimitado por las llaves { y } de la clase principal -en este caso `HolaJava`-

```
public class HolaJava { // llave de apertura de la clase

	// declaración del método
	public static void main(String[] args) {

		// implementación del método
		System.out.println("Hola Java!");
	}

} // llave de cierre de la clase
```

Sintaxis de declaración

La secuencia en la que se debe colocar los distintos elementos de la declaración de un método está estrictamente definida por la sintáxis de Java. Es como sigue:

`[<modificadorAcceso>] [static] <tipoReturn> nombreMetodo>([<Parametros>])`

Los elementos obligatorios en la declaración de un método son:

1. Tipo del valor de retorno del método **`<tipoReturn>`** que puede ser:
   - ➢ **`void`** si no hay **`return`**.
   - ➢ Cualquier tipo primitivo del lenguaje.
   - ➢ Cualquier tipo de dato definido por el programador: **`class`**, **`interface`** o **`enum`**.
   - ➢ Array de cualquier tipo.
2. Nombre del método **`<nombreMetodo>`**
3. Un par de paréntesis **`( )`**

Los elementos opcionales, que pueden no aparecer, son:

- ❖ Modificador **`static`**
- ❖ **`<modificadorAcceso>`** que puede ser: **`public`**, **`private`** o **`protected`**
- ❖ Lista de parámetros del método **`<parametros>`**

Firma o signatura

La signatura es el medio que utiliza el lenguaje para reconocer de manera inequívoca un método. La signatura está formada por dos elementos de su declaración:

- ❑ El nombre del método.
- ❑ La lista de sus parámetros.

Nombre del método

La manera de usar y ejecutar el código de un método es invocar o llamar al método por su nombre.

```
public class HolaJava {

	public static void main(String [] args) {

		// Llama a un método para hacer algo
		mostrarMensaje();
	}

	// Declara e implementa un método
	static void mostrarMensaje() {

		System.out.println( "Hola Java!");
	}
}
```

Estilo para los identificadores de métodos

El estilo estándar de Java recomienda que:

- El nombre de un método debe comenzar con una letra minúscula.
- Se utiliza la técnica *CamelCase*, es decir, los nombres compuestos utilizan mayúsculas intercaladas para la primera letra de cada una de las partes.
- Los nombres deben ser descriptivos del propósito, normalmente indican acciones, y por lo tanto se recomienda utilizar verbos o un verbo y un sustantivo.

Modificadores

Un modificador es una palabra clave del lenguaje Java que proporciona información adicional para el compilador. Los modificadores aplicables en la declaración de métodos son **`public`**, **`private`**, **`protected`** y **`static`**. Están relacionados con los aspectos de orientación a objetos de los mismos. Son tratados más ampliamente en el capítulo sobre las clases.

Los modificadores de acceso no tienen efecto si el método se invoca desde dentro de la misma clase y por lo tanto se pueden omitir en programas con una sola clase. Una declaración de método no puede tener más de un modificador de acceso.

Cuando un método lleva el modificador **`static`** en su declaración, significa que este método puede ser llamado por cualquier otro método sin estar ligado a un objeto; es lo más parecido que hay en Java a un subprograma de librería.

Implementación de métodos

Una vez que se ha declarado la cabecera de un método se puede escribir su implementación. **La implementación consiste en la escritura de las sentencias y las instrucciones de código que se ejecutarán cuando se invoque el método.** Este código debe colocarse en el cuerpo del método entre llaves. **Contiene las instrucciones y sentencias necesarias para cumplir un único fin funcional -Principio de Responsabilidad Única-.** También **debe ser pequeño en tamaño -no más de quince o veinte líneas de código útil-; y conseguir que el número de métodos**

necesarios, sea también pequeño.

El desarrollo de métodos con criterios de calidad del código es un ejercicio de equilibrio o compromiso entre cohesión, acoplamiento, tamaño y granularidad.

Cuerpo del método

Es el bloque de instrucciones que se encuentra entre las llaves inmediatamente después de la declaración del método.

```
[<modificadorAcceso>] [static] <tipoReturn>
<nombreMetodo>([<Parametros>]) {

      // ... aquí va la implementación, es el cuerpo del método ...

}
```

El trabajo efectivo que realiza un método se encuentra precisamente en el cuerpo del método.

```
// Declara e implementa un método
static void mostrarMensaje() {

      // Cuerpo del método
      System.out.println( "Hola Java!");
}
```

Es importante destacar que los métodos en Java no se pueden anidar, osea que dentro del cuerpo de un método no se puede declarar ni implementar otro método, sólo se puede llamar.

Variables de método

Las variables declaradas dentro del cuerpo de un método se llaman variables locales de método.

Es importante destacar un aspecto general del lenguaje Java: **Las variables son locales al bloque -delimitado por llaves- donde se declaran; y en ese contexto no pueden coexistir nombres repetidos.**

```
static void mostrarMensaje() {
      // variable local
      String mensaje = "Hola Java!";
      System.out.println(mensaje);
}
```

Llamadas a métodos

La invocación o llamada a un método implica la ejecución del código que se incluye en el

cuerpo del método o implementación.

Para invocar a un método hay que escribir exactamente el nombre del método seguido de paréntesis que encierran los parámetros previstos en la declaración terminando en **;**

```
<nombreMetodo>([<Parametros>]);
```

Transferencia del control en las llamadas a métodos

Cuando se invoca un método se produce un salto, del flujo de ejecución, al cuerpo del método llamado. Una vez se llega al final del método llamado, se devuelve el control de ejecución a la siguiente instrucción del punto de llamada.

Independencia entre la declaración y la llamada a un método

Java permite que dentro de una misma clase se invoque a un método que aparece declarado posteriormente en el código.

```
public static void main(String[] args) {
	// ..
	printLogo();		// ejecuta el método...
	// ..
}

static void printLogo()
	System.out.println("IES Ingeniero de la Cierva");
}
```

Se puede comprobar que aunque la invocación del método está antes de la declaración del método, el programa se compila y ejecuta sin ningún problema. En otros lenguajes de programación no se permite el uso de un método antes de su línea de su declaración.

Métodos con parámetros

A menudo, los métodos necesitan información adicional para adaptarse a una situación específica o personalizada de datos que dependen del contexto de uso. Para esto se utiliza una lista de parámetros entre paréntesis en la declaración del método:

```
public static <tipoReturn> <nombreMetodo>(<parametros>) {
   // cuerpo del método
}
```

`<parametros>` , es una **lista de declaraciones de variables separadas por comas, que se utilizarán como variables locales dentro del método** para algún fin.

La lista de parámetros de un método puede estar formada por cualquier cantidad y tipos de datos; primitivos y referencias. Un ejemplo puede ser:

```
public static void main(String[] args) {
	// ..
	String logo = "IES Ingeniero de la Cierva";
	printLogo(logo);		// de forma oculta se produce: lg = logo;
	// ..
}

static void printLogo(String lg) {
	System.out.println(lg); // muestra lo que llegue en lg
}
```

Cuando se declara un método con parámetros, el objetivo es que cada vez que se llame pueda cambiar el resultado en función de los datos de entrada. En otras palabras, el algoritmo que se describe en el método será único, pero el resultado final podrá ser múltiple, dependiendo de los datos proporcionados a través de la lista de parámetros.

Los parámetros permiten, que un método se adapte a diferentes situaciones de datos para un mismo proceso genérico; contribuyen a la reutilización y optimización del código.

Otro ejemplo de un método que cambia su comportamiento según sus parámetros de entrada es este: Un método recibe un entero, y en función de si el número es positivo, negativo o cero, se muestra en la consola - `Positivo`, `Negativo` o `Cero`.

```
static void muestraSigno(int numero) {
	if (numero > 0) {
		System.out.println("Positivo");
	}
	else {
		if (numero < 0) {
			System.out.println("Negativo");
		}
		else {
			System.out.println("Cero");
		}
	}
}
```

Ejemplo: Método con varios parámetros

Un método puede recibir varios parámetros, como en el siguiente ejemplo en el que se muestra el mayor de dos números.

```
static void muestraMayor(double num1, double num2) {
	double max = num1;
	if (num1 > num2) {
		max = num2;
	}
	System.out.println("Mayor: " + max);
}
```

En la lista de parámetros, no se admite la sintaxis abreviada de declaración; el siguiente código produce un error...

```
static void muestraMayor(double num1, num2) {   // ERROR...
	// ..
}
```

La llamada a un método con varios parámetros se realiza de la misma manera que un método sin parámetros. **Cuando son varios los parámetros, se debe proporcionar entre paréntesis la lista correspondiente**.

Parámetros actuales y parámetros formales

Cuando se declara un método, los elementos de la lista de parámetros, se llaman *parámetros formales*.

En la llamada, los parámetros que se proporcionan son los *argumentos* reales o *parámetros actuales*.

Los argumentos reales son asignados a los *parámetros formales* declarados.

En el ejemplo anterior, `num1` y `num2` son los *parámetros formales*; los argumentos o parámetros actuales serían `25` y `30` en el siguiente ejemplo de llamada al método:

```
//...
muestraMayor(25, 30);  // num1 = 25; num2 = 30
//...
```

Paso de argumentos por valor

Los parámetros actuales de cualquier tipo primitivo del lenguaje siempre son replicados como copia independiente al asignarse a los parámetros formales; cualquier cambio que se pueda producir dentro del método de esos datos no repercute de ninguna manera en los datos asociados a los parámetros actuales. Un ejemplo:

```
static void mostrarNumero(int num) {
```

```
        // Modifica el parámetro de tipo primitivo recibido
        num += 2;
        System.out.println("En mostrarNumero() se modifica, el parámetro
        recibido: " + num);
}
```

Al llamar al método :

```
public static void main(String[] args) {
   int dato = 3;
   mostrarNumero(dato);     // num = dato;  que  vale 3
   System.out.println("En main(), dato vale: " + dato);
}
```

Una vez que se llama al método *mostrarNumero*(dato) El *parámetro actual* dato con valor 3 se asigna y copia al *parámetro formal* num; la primera instrucción del método añade 2 a la variable num. Esto no afecta a la variable dato. Por lo tanto, el método *mostrarNumero*(dato) muestra el valor 5 y el método main () muestra el valor 5.

```
En mostrarNumero() se modifica, el parámetro recibido: 5
En main(), dato vale: 3
```

Expresiones como argumentos

Cuando se llama a un método, se pueden proporcionar expresiones como argumentos. Cuando se hace esto, Java calcula los valores de estas expresiones en tiempo de ejecución (y de ser posible en tiempo de compilación) sustituyendo la expresión por el resultado.

```
 mostrarSigno(2 + 3);
 double dato1 = 3;
 double dato2 = 2;
 mostrarMayor(dato1 * 5, dato2 * 2);
```

El resultado de la ejecución de estos métodos es la siguiente:

```
Positivo
Mayor: 15.0
```

Argumentos compatibles

Se pueden proporcionar *argumentos* compatibles con el tipo de dato declarado en la lista de *parámetros formales* del método.

Por ejemplo, si el *parámetro formal* declarado se espera que sea de tipo **`double`** , en la invocación del método se puede proporcionar un valor de tipo **`int`**. Se produce una conversión implícita o automática.

```
static void muestraNumero(double num) {
   System.out.println("El número  double: " + num);
}
public static void main(String[] args) {
   muestraNumero(5);
}
```

Si el método `muestraNumero()` se invoca con el literal `5` , que es el tipo predeterminado **`int`** , se convierte en el valor de punto flotante de doble precisión `5d`. Por lo tanto, transformado, este valor se pasa al método `muestraNumero()` .

Como se puede suponer; el resultado es:

```
El número  double: 5.0
```

Compatibilidad con expresiones

El resultado de una expresión que se pasa como argumento debe ser del mismo tipo que el tipo declarado para el *parámetro formal* o un tipo compatible.

Por ejemplo, si el parámetro requerido es de tipo **`double`**, se permite que el cálculo del valor de la expresión sea de tipo **`int`** .

```
public static void main(String[] args) {
   muestraNumero(3 + 2);
}
```

El resultado es:

```
El número  double: 5.0
```

Orden en la secuencia de argumentos

Los valores que se pasan a un método en su invocación deben ir en el mismo orden que los *parámetros formales* en la declaración. Esto está relacionado con la *signatura del método*.

Si se supone un método `mostrarNombreEdad()`, que recibe un `String` y un **`int`** para mostrar el nombre y la edad de una persona:

```
public class Persona {
      public static void main(String[] args) {
```

```
            mostrarNombreEdad(24, "Pepe"); // ERROR
      }

      static void mostrarNombreEdad(String nombre, int edad) {
            System.out.println("Soy " + nombre + ", " + edad + " años.");
      }
}
```

Si el compilador no encuentra un método cuya firma concuerde. Se produce un mensaje de error parecido a:

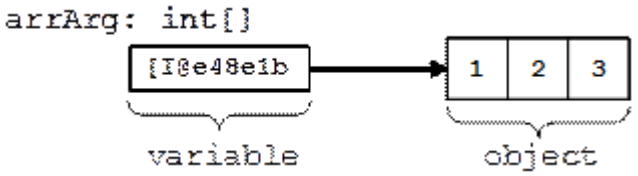
```
The method mostrarNombreEdad(int, String) in the type Persona is not
applicable for the arguments (String, int)
```

Cuando son varios los parámetros de un método, hay que tener en cuenta que deben mantener coincidencia completa en el número, tipo de dato y orden con la lista que se declaró.

Paso de argumentos por referencia

Cuando los parámetros implicados en el uso de métodos son de tipo referencia (tales como arrays), hay que tener mucho cuidado. **Los tipos de referencia constan siempre de dos partes, la variable o referencia propiamente dicha (una dirección de memoria) y el valor o valores referenciados**.

En el caso de los arrays se almacenan de la siguiente manera en la memoria:



Un *arrays* siempre utiliza el nombre de la variable con la que se declara para acceder a sus datos; esta variable es una referencia. Por lo tanto, **cuando se pasa un *argumento* de tipo *array*, el valor de la *variable referencia* (la dirección de memoria donde están sus elementos) se copia en el *argumento formal***.

Pero ¿Qué pasa con el objeto o elementos que lo forman? ¿Se copian?

Se puede comprobar en el siguiente ejemplo, que no se copian:

Un método que modifica el primer elemento de un array, poniéndolo con valor 5, y luego muestra todos los elementos:

```
static void  modificaArray(int[] arrArg {
       arrArg[0] = 5;
       System.out.print("En modificaArray() el arrArg es: ");
       System.out.println(Arrays.toString(arrArg));    //muestra arrArg
}
```

En `main()`:

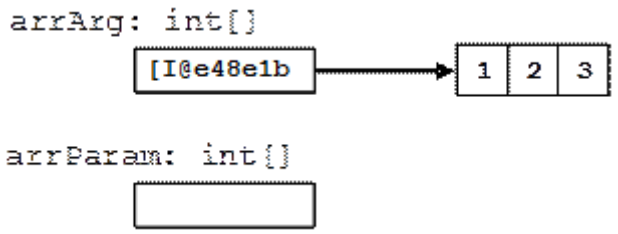
```
public static void main(String[] args) {
       int[] arrParam = new int[] { 1, 2, 3 };
       System.out.print("En main() antes de modificaArray(), los datos son:
       ");
       System.out.println(Arrays.toString(arrParam));   //muestra arrParam
       // llama al método
       modificaArray(arrParam);  //arrArg = arrParam;
       System.out.print("En main() después de modificaArray(), los datos
       son: ");
       System.out.println(Arrays.toString(arrParam));   //muestra arrParam
}
```

La salida del programa sería:

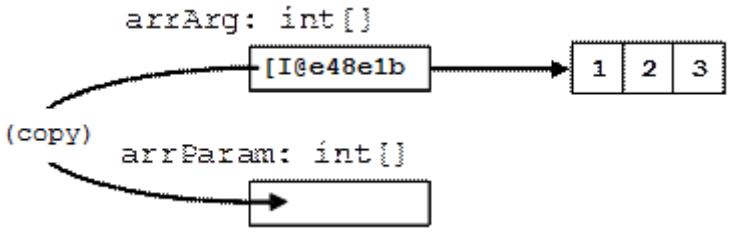
```
En main() antes de modificaArray(), los datos son: [1, 2, 3]
En modificaArray() el arrParam es:  [5, 2, 3]
En main() después de modificaArray(), los datos son: [5, 2, 3]
```

La razón de este resultado es que al pasar un argumento de tipo de referencia a un método, sólo se copia el valor de la variable que mantiene una referencia al objeto, pero no se hace una copia del objeto en sí.

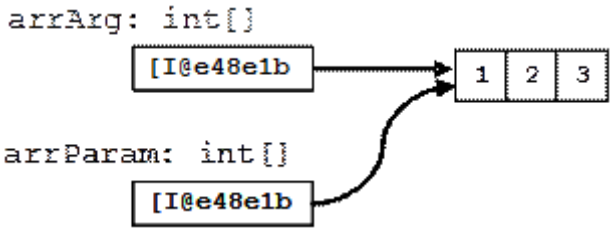
Antes de la invocación del método `modificaArray()`, el valor del parámetro `array` está indefinido y no guarda una referencia a un objeto concreto; vale **`null`**.



Durante una llamada a `modificaArray()`, el valor que se almacena en el argumento `arrArg`, copiar parámetro `arrParam`:



Por lo tanto, duplica la referencia a los elementos del array pero no los datos:



Este es el motivo por el que hay que tener cuidado, si el método llamado modifica el objeto pasado por referencia, puede afectar a la ejecución del código que sigue a la llamada del método. Esa es la gran diferencia entre un argumento de tipos primitivos y de referencia.

Múltiples datos con un solo argumento

Es posible proporcionar cualquier cantidad de datos del mismo tipo agregados en una estructura que se pasa por referencia; como se muestra en el ejemplo:

```
public class Libros {
	public static void main(String[] args) {
		double[] preciosLibros = new double[] { 3, 2.5 };
		muestraMontanteTotalLibros(preciosLibros);
	}

	static void muestraMontanteTotalLibros(double[] precios) {
		 double total = 0;
		 for (double p: precios) {	// para cada uno de los valores
		 del array
			total += p;
		 }
		 System.out.println("El montante total de libros es: " +
		 total);
	}
}
```

Esta manera de proporcionar múltiples datos a un método era así hasta la versión de Java 5, a partir de la cual se permite alternativamente la siguiente forma equivalente:

```
// ...
	public static void main(String[] args) {

		// implicitamente equivale precios = new double[] {3, 2.5}
		muestraMontanteTotalLibros(3, 2.5);

		muestraMontanteTotalLibros(3, 5.1, 10, 4.5);
	}
// ...
```

Si un método espera recibir un array de datos, es posible presentar una lista de valores que corresponden a los elementos con los que se cargará el array del parámetro formal.

Declaración de un número variable de parámetros

Es posible la declaración formal de métodos con un número variable de argumentos utilizando tres puntos suspensivos entre el tipo de dato y el nombre del parámetro como se muestra en el ejemplo:

```
public class SumaNumeros {
	public static void main(String[] args) {
		long sum = sumar(2, 5);
		System.out.println(sum)

		long sum2 = sumar(4, 0, -2, 12);
		System.out.println(sum2);

		long sum3 = sumar();
		System.out.println(sum3);
	}

	static long sumar(int ... numeros) {
		long sum = 0;
		for (int n : numeros) {
			sum += n;
		}
		return sum;
	}
}
```

salida:

```
7
```

```
14
0
```

El método `sumar()` suma cualquier serie de números proporcionada como argumento.

El parámetro de la declaración formal de arriba, que permite la presentación de un número variable de argumentos, es en realidad el nombre de **un array** del tipo especificado.

En la llamada a un método con número variable de argumentos, de un determinado tipo, es perfectamente válido proporcionar un array equivalente.

Si un método tiene parámetros convencionales y un parámetro de número variable, éste debe ser único e ir al final de la lista de parámetros formales.

```
static void metodo(int dato, String nombre, int ... x) {
	// ...
}
```

Si se pone la declaración del parámetro **`int ... x`**, del ejemplo en otra posición:

```
static void metodo(int dato, int ... x, String nombre) {
	// ...
}
```

El compilador mostrará el siguiente mensaje de error:

```
The variable argument type int of the method doSth must be the last parameter
```

Resultado de un método

En ocasiones un subprograma resulta más operativo si proporciona un resultado de su proceso… Es lo que tradicionalmente se llaman funciones.

Para este fin los métodos de Java puede devolver un resultado utilizando la palabra reservada **`return`**.

Declaración de un método con valor de retorno

En la sintaxis de declaración ya conocida:

`[<modificadorAcceso>] [static] <tipoReturn> nombreMetodo>([<Parametros>])`

Hay una sección relacionada con el tipo del valor de retorno del método **`<tipoReturn>`** que indica que el método devolverá un dato.

Así es como se vería un método que calcula el área de un cuadrado y devuelve un resultado de tipo **double**:

```
static double calcularSuperficieCuadrado(double lado) {
      return lado * lado;
}
```

Uso del valor de retorno

Cuando se ejecuta un método y devuelve un valor de retorno, Java pone este valor en el lugar en el que se está llamando al método y continúa la ejecución. En consecuencia, este valor de retorno, se puede utilizar en el sitio donde se llamar para diferentes propósitos.

Asignar a una variable

Se puede asignar el resultado del método a una variable del tipo apropiado:

```
// ...
double area = calcularSuperficieCuadrado(5);
```

Utilización en expresiones

Una vez que el método devuelve un resultado, pode ser utilizardo en expresiones.

```
// ...
double areaTotal = calcularSuperficieCuadrado(5)
                   + calcularSuperficieCuadrado(6);
```

Aplicación como parámetro a otro método

Se puede proporcionar el resultado de un método como argumento para otro método:

```
// ...
System.out.println(calcularSuperficieCuadrado(5));
```

En este ejemplo, primero se llama al método `println()` pasándole como argumento `calcularSuperficieCuadrado()`. Una vez que el método `calcularSuperficieCuadrado()` se ejecuta, devuelve el resultado que Java utiliza en la invocación del método `println()`, con algo equivalente a:

```
System.out.println(25);
```

Tipo de valor devuelto

Como se ha indicado, el resultado que devuelve un método puede ser:

- **void** si no hay **return**.
- Cualquier tipo primitivo del lenguaje.

- ➢ Cualquier tipo de dato definido por el programador: **`class`**, **`interface`** o **`enum`**.
- ➢ Array de cualquier tipo. Por ejemplo, si tomamos el ejemplo de un método que calcula una cara cuadrada en lugar del valor impreso en la consola, el método puede devolver como resultado de ello.

return

Para hacer que un método devuelva un dato, en su implementación, debe utilizarse la palabra reservada **`return`** seguida por el resultado del método. Por ejemplo:

```
static int multiplicar(int num1, int num2) {
    int result = num1 * num2;
    return result;
}
```

En este método, después de la multiplicación, debido al **`return`**, se devolverá el resultado de la ejecución del proceso -un valor de tipo **`int`**-

Retorno de un tipo compatible

El resultado devuelto por un método puede ser de un tipo compatible (que se puede convertir implícitamente) al tipo del valor de retorno declarado .

Por ejemplo, se puede modificar el ejemplo anterior para que el tipo retorno sea **`double`**, no **`int`** y mantener el resto del código:

```
static double multiplicar(int num1, int num2) {
      int result = num1 * num2;
      // Java realiza una conversión implícita semejante a:
      // return (double) result;
      return result;
}
```

Retorno de una expresión

La sentencia return admite una expresión cualquiera a condición de que su resultado sea compatible o convertible en un valor del tipo declarado. Por ejemplo:

```
static int multiplicar(int num1, int num2) {
    return num1 * num2;
}
```

En este caso, la expresión `num1 * num2` se calculará y su resultado será devuelto.

Características de return

La palabra reservada **`return`** hace dos operaciones:

- Termina la ejecución del método y devuelve el control de ejecución al punto de llamada.
- Devuelve el resultado del método.

En relación con la primera operación relacionada con **`return`**, hay que destacar que debido a que termina la ejecución del método no puede haber ninguna instrucción después del **`return`** de un método. Por ejemplo:

```
static int multiplicar(int num1, int num2) {
	return num1 * num2;
	// Inaccesible
	if (num1 == 3) {
	}

}
```

En este caso, la compilación fallará. Para las líneas que hay después de **`return`** , el compilador mostrará un mensaje de error similar al siguiente:

```
Unreachable code
```

Cuando el método tiene un tipo de valor devuelto **`void`**, se puede utilizar **`return`**, sólo para terminar el método sin valor de retorno. En este caso el uso de **`return`** es para la terminación del proceso:

```
static void muestraNumeroPositivo(int num) {
    if (num <= 0) {
        // Si el número es negativo, termina el método
        return;
    }
    System.out.println(num);
}
```

Otro detalle que hay que destacar de **`return`** es que se puede aparecer en varios lugares en un método si se cumplen varias condiciones.

En el ejemplo se define un método que recibe dos números como parámetros y los compara:

```
static int comparar(int num1, int num2) {
	if (num1 > num2) {
		return 1;
	}
```

```
        if (num1 < num2) {
                return -1;
        }
        return 0;
}
```

Diseño por contrato

Diseñar por contrato es una técnica de programación -aplicable al diseño de métodos- relativamente difundida. La idea es simple: **Se especifican las condiciones de lo que se recibe como argumentos de entrada y de lo que se tiene que devolver como resultado**. Estos términos constituyen el *contrato entre las partes* y a él hay que atenerse en caso de errores para determinar responsabilidades.

En términos de lógica matemática, se suele hablar de:

- **precondición**
  Lo que debe ser cierto cuando empieza a ejecutarse un fragmento de código.
- **postcondición**
  Lo que debe ser cierto cuando termina la ejecución un fragmento de código.
- **invariante** (de bucle)
  lo que debe ser cierto mientras un fragmento de código se está ejecutando; típicamente interesa centrar condiciones que deben satisfacerse en todas y cada una de las iteraciones de un bucle.

El fallo de una *precondición* implica, necesariamente, que el *usuario-cliente* del código no está haciendo un uso según lo contratado. Si, por ejemplo, se desarrolla un programa para calcular la raíz cuadrada de un número, es razonable exigir, como precondición, que el número sea positivo.

El fallo de una *postcondición* quiere decir que no se cumple con el compromiso adquirido por la pieza de código afectada. Si, por ejemplo, se desarrolla un programa para calcular la raíz cuadrada de un número con una cierta precisión, se exigirá, a la salida, que el resultado multiplicado por sí mismo, difiera del dato de entrada en menos que la *precisión contratada*.

El fallo de un invariante de bucle significa que el algoritmo se ha perdido. Por ejemplo el caso de estar buscando una palabra en un diccionario por el algoritmo de *búsqueda binaria*. El algoritmo consiste en abrir el diccionario por la mitad y determinar en qué mitad está la palabra, y así sucesivamente con la mitad, de la mitad, de la mitad, etc. **En todo momento debe ser cierto que la palabra que se busca esté en la parte del diccionario donde se está buscando**; **es decir, que sea posterior a la primera palabra y anterior a la última de parte donde se está buscando.**

Validación de datos

Un método cliente - que delega en otros-, bajo la técnica de programación por contrato, siempre debe proporcionar datos válidos a los métodos que utiliza y debe gestionar la posibilidad de que los datos que maneja sean erróneos según el contrato que debe satisfacer; situación en la que debe chequear y rechazar los datos si es necesario. Es una cadena de responsabilidades.

El método que tenga la responsabilidad de proporcionar datos válidos a otro es el que debe implementar código para comprobar la posible situación anómala y resolverla si es factible; nunca con asertos. En el peor de los casos debe propagar el error.

Los asertos se usan cuando algo, ni debe ni puede ocurrir -teóricamente-. Cuando la teoría dice que un programa es correcto; pero la práctica lo desmiente, es cuando los asertos lanzan su excepción.

Ejemplo con una buena asignación de responsabilidades:

```
import java.util.Scanner;

public class FibonacciConAsertos {
	final static int LIMITE_INFERIOR = 0;
	final static int LIMITE_SUPERIOR = 161;

	public static void main(String[] args) {
		System.out.println("Número de Fibonacci del término (n).");
		System.out.print("Introduce n = ");
		int numero = new Scanner(System.in).nextInt();

		if (numero >= LIMITE_INFERIOR && numero <= LIMITE_SUPERIOR) {
			System.out.println(fibonacci(numero)
			+ "\nEs el número de Fibonacci para n = " + numero);
		}
		else {
			System.out.println("Fuera de rango,
				sólo se admiten valores entre: "
				+ LIMITE_INFERIOR + " y " + LIMITE_SUPERIOR);
		}
	}

	/**
	 * Calcula el término n de la sucesión de Fibonacci.
	 * @param n
	 * @return el término n
	 */
	private static long fibonacci(int n) {
```

```
            assert n >= LIMITE_INFERIOR: "Número negativo";
            assert n <= LIMITE_SUPERIOR: "No calculable";

            long terminoActual = n;
            long terminoMenos1 = 1;
            long terminoMenos2 = 0;

            for (int i = 1; i < n; i++) {
                  terminoActual = terminoMenos1 + terminoMenos2;
                  terminoMenos2 = terminoMenos1;
                  terminoMenos1 = terminoActual;
            }
            return terminoActual;
      }

} //class
```

Fuentes y bibliografía

- NAKOV, S. *Fundamentos de programación con Java*. Ed. Faber, Veliko Tarnovo, 2009. [en linea][traducción automática del búlgaro] http://www.introprogramming.info/intro-java-book/
- CARRERES, J. *Manual de Java*. [en línea] http://www.oocities.org/collegepark/quad/8901/indice.html
- MAÑAS, J. A. *Laboratorio de programación. Material de estudio. Apuntes.* [en línea] http://www.lab.dit.upm.es/~lprg/material/apuntes/index.html