Funciones y procedimientos

Objetivos

- Utilizar correctamente el concepto de rutina.
- Conocer el concepto de reutilización del código.
- Crear código reutilizable.
- Saber reconocer el concepto de función, argumento y valor.
- Conocer el concepto de procedimiento y pasaje de parámetros.

Contenidos

Definición de sub-algoritmo o rutina. Reutilización. Razones para definir rutinas. Abstracción. Cohesión. Acoplamiento. Ocultamiento de la información. Modularización.

Concepto de función. Argumento y valor. Concepto de procedimiento. Pasaje de parámetros por valor y por referencia. Variables locales. ¿Qué es main()? Diferencia entre definir e invocar una función/procedimiento. Noción de la diferencia entre función (paradigma estructurado) y método (POO).

Ambito de las variables

En Java, no todas las variables son accesibles desde cualquier lugar, sino que depende del contexto en el cual han sido declaradas. A esto se lo conoce como ámbito (en inglés, “scope”).

El siguiente programa muestra lo que ocurre cuando se intenta acceder a una variable declarada dentro de un bloque de selección:

```
1   package prueba;
2   import java.util.Scanner;
3   public class Prueba {
4           public static void main(String[] args) {
5                   Scanner entrada = new Scanner(System.in);
6                   int num;
7                   System.out.print("Ingresá un número: ");
8                   num = entrada.nextInt();
9                   if (num> 0) {
10                          String cadena = "Es positivo";
11                  }
12                  System.out.println(cadena); // No compila
13          }
14  }
```

Esto ocurre porque **cadena** fue declarada dentro del bloque if, por lo que solo es accesible desde dentro del bloque en cuestión y otros que pudiera haber dentro. Esto aplica para variables que se declaren dentro de cualquier bloque if, else, switch, for, while o do…while.

Para solucionar el problema anterior, tendríamos que haber declarado la variable **cadena** fuera del bloque if:

Al haber sido declarada al principio de la función main, la variable cadena es accesible desde cualquier lugar dentro del método.

Modularización

El concepto de modularización es uno de los más importantes de la programación.

Existen programas complejos, que realizan muchas y variadas operaciones de cómputo, quizá varias veces con diferentes datos. Es mucho más fácil diseñar, codificar, depurar y mantener un programa creado a través de la unión de pequeñas piezas de código independientes entre sí que intentar tratarlo como un todo.

La idea a partir de aquí es realizar programas de manera modularizada, es decir, separando cada proceso en bloques independientes que luego “unirán fuerzas” para lograr el objetivo inicial.

Conocer y comprender esta metodología, te permitirá no tan solo tener un código mucho más legible y ordenado, sino también ahorrar mucho tiempo y esfuerzo mental, dado que muchas veces se pueden reutilizar bloques ya testeados de otros programas creados con anterioridad, sin tener que volver a pensarlos.

Procedimiento

Un procedimiento es un bloque de código que puede o no recibir datos de entrada, pero que no retorna resultados. Su objetivo es permitir "sacar factor común" de instrucciones que se repitan frecuentemente en el código, trasladando en cierto momento el flujo hacia el procedimiento, y volviendo al punto desde donde se invocó cuando se terminan de ejecutar las instrucciones dentro de él.

Los procedimientos se dividen en dos partes: **la definición y la invocación.**

La **definición** de un procedimiento es justamente crearlo. Ponerle nombre, establecer qué datos espera recibir y escribir las instrucciones que deberían realizarse. La definición de un procedimiento se realiza una sola vez. El código que se haya definido en un procedimiento no va a ser ejecutado hasta que no se haga un pedido exclusivo.

La **invocación** de un procedimiento permite enviarle los datos necesarios al mismo, para que ejecute las operaciones que hay en su definición y, por tanto, provea el resultado esperado. La invocación de un procedimiento se realiza cada vez que sea necesario.

Veamos un ejemplo concreto con poca utilidad práctica pero que ilustra perfectamente el concepto anterior. El siguiente programa imprime por consola la letra original de la canción "We Will Rock You", de Queen:

```
1   package prueba;
2
3
4   public class Prueba {
5           public static void main(String[] args) {
6                   System.out.println("Buddy, you're a boy, make a big noise");
7                   System.out.println("Playing in the street, gonna be a big man some day");
8                   System.out.println("You got mud on your face, you big disgrace");
9                   System.out.println("Kicking your can all over the place");
10                  System.out.println(""); // Línea en blanco
11                  System.out.println("We will, we will rock you");
12                  System.out.println("We will, we will rock you");
13                  System.out.println(""); // Línea en blanco
14                  System.out.println("Buddy, you're a young man, hard man");
15                  System.out.println("Shouting in the street, gonna take on the world some
16  day"); System.out.println("You got blood on your face, you big disgrace");
17                  System.out.println("Waving your banner all over the place");
18                  System.out.println(""); // Línea en blanco
19                  System.out.println("We will, we will rock you");
20                  System.out.println("We will, we will rock you");
21                  System.out.println(""); // Línea en blanco
22                  System.out.println("Buddy, you're an old man, poor man");
23                  System.out.println("Pleading with your eyes, gonna make you some peace
24  some day"); System.out.println("You got mud on your face, you big disgrace");
25                  System.out.println("Somebody better put you back into your place");
26                  System.out.println(""); // Línea en blanco
27                  System.out.println("We will, we will rock you");
28                  System.out.println("We will, we will rock you");
29                  System.out.println(""); // Línea en blanco
30                  System.out.println("We will, we will rock you");
31                  System.out.println("We will, we will rock you");
32          }
    }
```

Habrás notado que tanto en esta como en varias otras canciones, hay algo que siempre se repite: el estribillo.

La idea entonces es no repetir código de forma redundante, sino buscar la manera de definir por única vez cómo mostrar el estribillo y luego reemplazar las ocurrencias de este por una invocación a un procedimiento llamado **mostrarEstribillo().**

Definición de un procedimiento

Las definiciones de los procedimientos se escriben fuera de la función main() pero dentro de la clase Prueba, como ilustra el siguiente código:

```
1  package prueba;
2
3  public class Prueba {
4          public static void main(String[] args) {
5                  // Todo el código hasta ahora iba obligatoriamente aquí
6          } // Llave de cierre de main
7          // Aquí se insertan las definiciones de los procedimientos
8  } // Llave de cierre de la clase
```

La sintaxis básica para definir un procedimiento es la siguiente:

```
static void <nombre_procedimiento> (<tipo><nom>, <tipo><nom>, ...) {
        <instrucciones>
}
```

Donde:

La palabra **static** es obligatoria. Es una palabra que cobra sentido cuando se habla del paradigma orientado a objetos.

void es un tipo de dato especial que significa vacío. Indica que el procedimiento no devuelve ningún valor.

<nombre_procedimiento> es el nombre que se le dará al procedimiento, siguiendo las mismas reglas de nomenclatura de las variables.

Los paréntesis son obligatorios.

El **par <tipo>** y **<nom>** declaran que se recibe un dato de nombre <nom> cuyo tipo de dato es <tipo>. A ese dato se lo denomina parámetro. Si un procedimiento recibe más

de un parámetro, se separa cada par con comas. Si un procedimiento no recibe parámetros, los paréntesis quedan vacíos.

La cantidad, orden y tipos de los parámetros dependen del criterio del programador.

Las llaves de apertura y cierre delimitan las instrucciones del procedimiento.

<instrucciones> son las instrucciones que se ejecutarán cuando se invoque al procedimiento.

Por convención, los procedimientos se suelen nombrar de manera tal que describan su funcionamiento. La utilización de verbos es bastante conveniente, pues los procedimientos realizan acciones.

Vamos a crear un procedimiento llamado **mostrarEstribillo()**, sin parámetros, que establezca lo que su propio nombre indica:

```
1   package prueba;
2
3   public class Prueba {
4           public static void main(String[] args) {
5                   // Toda la canción va escrita aquí
6           }
7
8           static void mostrarEstribillo () {
9                   for (int i = 0; i < 2; i++) {
10                          System.out.println("We will, we will rock you");
11                  }
12          }
13  }
```

Invocación de un procedimiento

Para realizar las instrucciones listadas en un procedimiento, es necesario invocarlo. La sintaxis básica para invocar un procedimiento es la siguiente:

```
<nombre_procedimiento>(<arg>, <arg>, ...)
```

Donde:

<nombre_procedimiento> es el nombre del procedimiento que se desea invocar. Debe haber sido definido previamente.

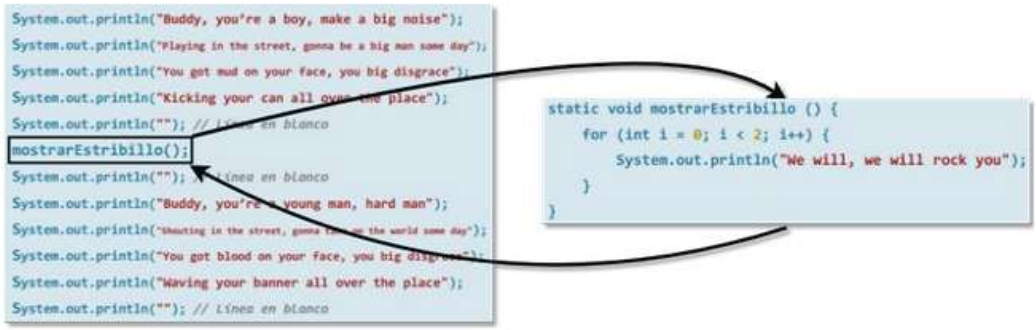
Los paréntesis son obligatorios.

<arg> es el dato requerido por el procedimiento que se desea invocar. En este contexto, se denomina argumento. Si un procedimiento requiere más de un argumento, se separan con comas. Si un procedimiento no requiere argumentos, los paréntesis quedan vacíos.

El siguiente código muestra la letra de la canción, utilizando invocaciones a **mostrarEstribillo()** donde corresponda:

```
1  package prueba;
2
3  public class Prueba {
4          public static void main(String[] args) {
5                  System.out.println("Buddy, you're a boy, make a big noise");
6                  System.out.println("Playing in the street, gonna be a big man some day");
7                  System.out.println("You got mud on your face, you big disgrace");
8                  System.out.println("Kicking your can all over the place");
9                  System.out.println(""); // Línea en blanco
10                 mostrarEstribillo();
11                 System.out.println(""); // Línea en blanco
12                 System.out.println("Buddy, you're a young man, hard man");
13                 System.out.println("Shouting in the street, gonna take on the world some
14 day"); System.out.println("You got blood on your face, you big disgrace");
15                 System.out.println("Waving your banner all over the place");
16                 System.out.println(""); // Línea en blanco
17                 mostrarEstribillo();
18                 System.out.println(""); // Línea en blanco
19                 System.out.println("Buddy, you're an old man, poor man");
20                 System.out.println("Pleading with your eyes, gonna make you some peace
21 some day");    System.out.println("You got mud on your face, you big disgrace");
22                 System.out.println("Somebody better put you back into your place");
23                 System.out.println(""); // Línea en blanco
24                 mostrarEstribillo();
25                 System.out.println(""); // Línea en blanco
26                 mostrarEstribillo();
27         }
28
29         static void mostrarEstribillo () {
30                 for (int i = 0; i < 2; i++) {
31                         System.out.println("We will, we will rock you");
32                 }
33         }
```

La siguiente imagen ilustra lo que el programa realiza:



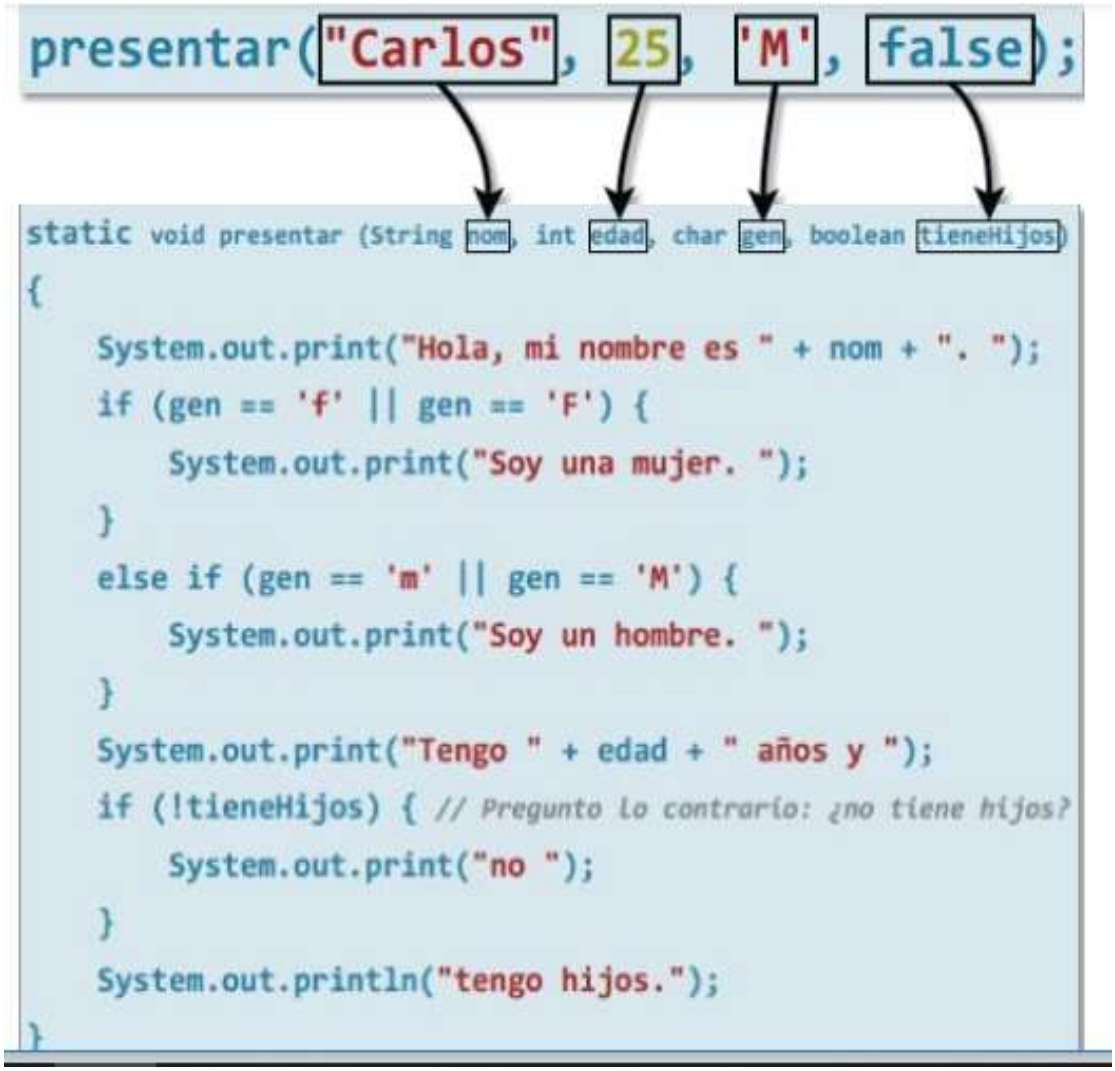
La salida del programa muestra exactamente lo mismo, pero el código es más "mantenible". Suponé que deseás agregar un punto final a cada uno de los "We will, we will rock you". Solo tenés que hacer el cambio en el procedimiento mostrarEstribillo(). De la manera anterior (sin usar procedimientos) tendrías que haber puesto un punto por cada una de las ocho veces que se repite la salida de la cadena.

Procedimiento con parámetros

El siguiente ejemplo muestra cómo realizar un procedimiento que reciba parámetros. El mismo elabora y escribe una presentación en la consola de acuerdo con los datos del usuario que provienen como parámetros, los cuales son el nombre, la edad, el género y si tiene o no hijos. Una vez definido el procedimiento, a través de la función main() haremos algunas pruebas con datos constantes, para ver los resultados:

```
1   package prueba;
2
3   public class Prueba {
4           public static void main(String[] args) {
5                   presentar("Carlos", 25, 'M', false);
6                   presentar("María", 46, 'f', true);
7                   presentar("Luis", 14, 'X', false);
8           }
9           static void presentar (String nom, int edad, char gen, boolean tieneHijos)
10          {
11                  System.out.print("Hola, mi nombre es " + nom + ". "); if (gen == 'f' || gen == 'F')
12  {
13                          System.out.print("Soy una mujer. ");
14                  }
15                  else if (gen == 'm' || gen == 'M') {
16                          System.out.print("Soy un hombre. ");
17                  }
18                  System.out.print("Tengo " + edad + " años y ");
19                  if (!tieneHijos) { // Pregunto lo contrario: ¿no tiene hijos?
20                          System.out.print("no ");
21                  }
22                  System.out.println("tengo hijos.");
23          }
24  }
```

La siguiente imagen ilustra lo que el programa realiza:



El procedimiento presentar() no tiene que conocer el origen de los valores de sus parámetros. Pudieron provenir de variables, ser constantes o introducidos por el usuario. No es relevante. El procedimiento tan solo captura los datos que se enviaron cuando se lo invocó, les asigna sus propios nombres y trabaja con ellos.

Es muy importante que a la hora de invocar un procedimiento se respete su "firma", es decir, el orden, el tipo y la cantidad de los argumentos que espera recibir. Así, en el ejemplo anterior, el procedimiento **presentar()** esperaba recibir los siguientes parámetros (los nombres para cada uno quedan a criterio del programador):
(String nom, int edad, char gen, boolean tieneHijos)
Por lo tanto, cuando se invoque al procedimiento, se debe respetar tal firma:
("Carlos", 25, ´M´, false)
("María", 46, ´F´, true)
("Luis", 14, ´X´, false)

Cualquier error, ya sea por tipos incompatibles, orden no respetado o cantidad no coincidente, produce que el programa no compile.

Funciones

Una función es un bloque de código que puede o no recibir datos de entrada, pero que retorna un resultado. Su objetivo es permitir hacer cálculos precisos mediante sus parámetros de entrada que resultarán en un dato de retorno.

Definición e invocación de una función

La sintaxis básica para definir una función es la siguiente:

```
static<tipo><nombre_función> (<tipo><nom>, <tipo><nom>, ...) {
      <instrucciones>return<valor>;
}
```

Donde:

La palabra **static** es obligatoria. Es una palabra que cobra sentido cuando se habla del paradigma orientado a objetos.

<tipo> es el de tipo de dato que retornará la función.

<nombre_función> es el nombre que se le dará a la función, siguiendo las mismas reglas de nomenclatura que las variables.

Los paréntesis son obligatorios.

El **par <tipo>** y **<nom>** declaran que se recibe un dato de nombre <nom> cuyo tipo de dato es <tipo>. A ese dato se lo denomina parámetro. Si una función recibe más de un parámetro, se separa cada par con comas. Si una función no recibe parámetros, los paréntesis quedan vacíos.

La cantidad, orden y tipos de los parámetros dependen del criterio del programador.

Las llaves de apertura y cierre delimitan las instrucciones de la función.

<instrucciones> son las instrucciones que se ejecutarán cuando se invoque a la función.

La palabra **return** indica que la función retornará un dato. En cuanto se ejecute la instrucción return, la función finaliza y el código retorna al punto desde donde se la invocó. Generalmente es la última instrucción del bloque.

<valor> es el valor que la función devolverá, cuyo tipo debe coincidir con él.

<tipo> a devolver en la definición.

Por convención, las funciones se suelen nombrar de manera tal que describan su funcionamiento. La utilización de verbos es bastante conveniente, pues las funciones realizan acciones para hacer sus cálculos.

Una función que no lleve la instrucción **return** o que tenga un camino donde la ejecución pudiera finalizar sin pasar por un **return** genera un error.

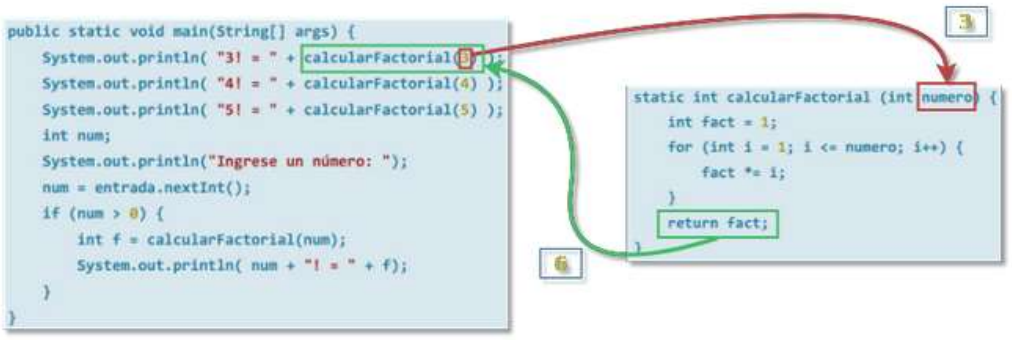
En el siguiente ejemplo, creo la función **calcularFactorial()**, que recibe un número entero como parámetro y devuelve el factorial del número:

```
1  package prueba;
2  import java.util.Scanner;
3  public class Prueba {
4          public static void main(String[] args) {
5                  System.out.println( "3! = " + calcularFactorial(3) );
6                  System.out.println( "4! = " + calcularFactorial(4) );
7                  System.out.println( "5! = " + calcularFactorial(5) );
8                  Scanner entrada = new Scanner(System.in);
9                  int num;
10                 System.out.print("Ingrese un número: ");
11                 num = entrada.nextInt();
12                 if (num> 0) {
13                         int f = calcularFactorial(num);
14                         System.out.println(num + "! = " + f);
15                 }
16         }
17         static int calcularFactorial(int numero) {
18                 int fact = 1;
19                 for (int i = 1; i <= numero; i++) {
20                         fact *= i;
21                 }
22                 return fact;
23         }
24 }
```

En las líneas 5, 6 y 7 se invoca a la función **calcularFactorial()** con valores constantes. En la línea 13 se invoca con un número entero introducido por el usuario, el cual debe ser mayor que 0, pues no existe el factorial de un número negativo.

Los nombres de las variables que son argumentos enviados en la invocación de una función no tienen por qué coincidir con los nombres declarados como parámetros en la definición de una función.
En la línea **13** del ejemplo anterior, se envía como argumento el valor de la variable **num**.
En la línea **17** se recibe ese valor como parámetro, el cual pasa a denominarse **numero**.
La función **calcularFactorial()** no tiene que conocer si el valor del parámetro provino de una variable, constante o un valor introducido por el usuario. Tan solo captura los datos que se enviaron en la invocación y les asigna sus propios nombres.

La siguiente imagen ilustra lo que el programa realiza:



Es muy importante que a la hora de invocar una función se respete su "firma", es decir, el orden, el tipo y la cantidad de los parámetros que espera recibir, al igual que en los procedimientos.

A partir de aquí, apelaremos a la palabra **función** para referirnos tanto a aquellos módulos que devuelven datos como aquellos que no.
Podría decirse que de todo procedimiento en realidad es una función, que retorna **void**.

Variables locales

Una función no deja de ser una pequeña parte de un programa, completamente independiente, que será invocada por otra que la necesite. En toda función se pueden declarar variables, leer y/o escribir datos en la consola, utilizar estructuras de control de flujo, etc.

Se trata de una "caja negra" independiente de su entorno, por lo tanto, todas las variables adicionales que se pudieran llegar a declarar dentro de ella son variables locales a la función. Solo pertenecen dentro del contexto de la función (delimitado por las llaves de apertura y cierre) y no están disponibles fuera.

La siguiente función **calcularFactorial()**:

```
static int calcularFactorial(int numero) {

    int fact = 1;

    for (int i = 1; i <= numero; i++) { fact *= i;

    }

    return fact;

}
```

Hace uso de una variable adicional de tipo int llamada fact. Dicha variable es local de la función y no es accesible fuera de ella.

Una variable local no puede tener el mismo nombre que un parámetro.

Diferencias entre procedimiento, función y método

Como habrás visto, un procedimiento se maneja igual que una función, con la diferencia de que un **procedimiento** no devuelve nada (tipo de retorno **void**) y una **función** devuelve (instrucción **return**) un dato. Aunque podemos generalizar y decir que todo procedimiento es una función, con la particularidad de que no devuelve datos.

En la programación estructurada queda bien claro el concepto de función. Pero como Java es un lenguaje orientado a objetos, en dicho paradigma tanto los datos como las funciones se encuentran dentro de una unidad llamada objeto, el cual se instancia a partir de una clase.

Si bien las funciones dentro de un objeto siguen los mismos conceptos que en el paradigma estructurado, en tal contexto se las llama **métodos**.

Para resumir:

- Una función en el paradigma estructurado existe por sí sola y es independiente de los datos.
- Un método en el paradigma orientado a objetos pertenece a un objeto en particular.
- Por ejemplo, nextInt() es un método del objeto entrada de tipo Scanner, que permite devolver un entero ingresado en la consola. En la clase Math, existe un método llamado sqrt() que permite realizar una raíz cuadrada.
- Para convenir, a las operaciones que se invocan provistas por Java a través de sus clases y objetos las llamaré como corresponde: métodos. A las operaciones que sean definidas por mí las llamaré funciones, aunque en realidad se trate de métodos estáticos de la clase que has creado (por defecto, los ejemplos que vengo trabajando se aplican sobre una clase llamada Prueba).

Acerca de main()

Habiendo visto algunos conceptos básicos, ahora podrás entender qué significa tener definido por obligación el siguiente bloque dentro de la clase, en los ejemplos, llamada Prueba:

```
public class Prueba {
    public static void main(String[] args) {
        // Tu código va aquí
    }
}
```

Dentro de la clase Prueba, excepto por la palabra **public**, que cobra sentido en la programación orientada a objetos, el resto luce como una definición de una función. De hecho, lo es.

Desde el primer programa que imprimía por consola un "Hola Mundo" hasta los últimos ejemplos vistos, todo tu código debía ir dentro de main().

En realidad, main() (en inglés, "principal") es un método de la clase Prueba, aunque nosotros lo llamaremos función, para adecuarnos al paradigma estructurado que estamos intentando emular en Java.

Pero, si el bloque anterior es simplemente la definición de lo que debe hacer la función main(), y sabiendo que una función no realiza sus acciones hasta que sea invocada, ¿cómo es que el código se ejecuta?

Es la JVM quien invoca a la función main() automáticamente al hacer ejecución del programa. Es el punto de entrada de toda aplicación Java, desde la más simple hasta la más compleja. Todo nace allí.

Se la declara con tipo de retorno void porque no devuelve datos. Recibe un único parámetro, que se trata de un arreglo de String llamado por defecto args (podrías cambiar el nombre sin problemas) Veremos más adelante el concepto de arreglo.

Hasta aquí nunca has usado el parámetro args, pues no lo necesitás. Eso comprueba que una función puede definir que recibe ciertos parámetros, pero nunca hacer uso de ellos. Por supuesto, en tus propias definiciones de funciones esto no tendría sentido práctico. El caso de main() es particular.

Toda aplicación comienza y termina en la función **main()**. Ahora que conoces cómo modularizar un programa en partes más pequeñas puede que pierdas un poco el seguimiento del flujo de las instrucciones. Vale la aclaración.

```
1  package prueba;
2
3  public class Prueba {
4          public static void main(String[] args) {
5                  System.out.println("En main, justo antes de invocar");
6                  System.out.println("5! = " + calcularFactorial(5));
7                  System.out.println("En main, luego de la invocación");
8          }
9          static int calcularFactorial (int numero) {
10                 int fact = 1;
11                 for (int i = 1; i <= numero; i++) {
12                         fact *= i;
13                 }
14                 return fact;
15         }
16 }
```

Pasaje de parámetros por valor y referencia

Habrás visto que hay casos en los que ciertas funciones o procedimientos requieren de parámetros para funcionar, por lo tanto, cuando se los invoca es necesario respetar esa "firma" y enviar los argumentos correspondientes.

Lo que es importante aclarar es que, a la hora de invocar una función, enviando como argumento el valor de una variable, lo que hace Java es copiar el valor de dicha variable en el parámetro correspondiente en la definición de la función. Si dentro de la definición de la función, el parámetro cambia su valor, tal cambio no se ve reflejado fuera de la misma, porque se está trabajando con una copia. Este comportamiento se denomina pasaje de parámetros por valor y es de la única manera en la que Java trabaja.

Otros lenguajes de menor nivel, como C, permiten hacer pasaje de argumentos por referencia. En vez de evitar una copia del valor de la variable, se envía la dirección de memoria de la variable (conocida como puntero o apuntador), por lo que la función podría acceder directamente a la variable y modificar su valor para todo el programa.

El siguiente código te ayudará a entender este concepto:

```
1   package prueba;
2
3   public class Prueba {
4           public static void main(String[] args) { int x = 5;
5           System.out.println("Muestro x antes de invocar: " + x);
6           duplicarValor(x);
7           System.out.println("Muestro x luego de invocar: " + x);
8           }
9
10          static void duplicarValor(int numero) {
11                  numero *= 2;
12          }
13  }
```

Ambas salidas muestran el valor 5. Por lo que queda demostrado que cuando se invoca en la línea 6 a la función **duplicarValor()** con el argumento x, se está enviando una copia del valor de x. En la definición de la función **duplicarValor()**, se recibe tal valor como parámetro con el nombre numero (pudo también haberse llamado x, son independientes, pero no quiero confundir). A pesar de que en la línea 11, se reemplaza a la variable numero con el doble de su valor, tal parámetro es independiente de la variable local x en la función main().

Cuestiones de diseño

Cohesión

Se denomina **cohesión** a la medición de cuán mínimo es el proceso en un módulo de software. En un buen diseño de soluciones, las funciones deben ser altamente cohesivas, es decir, sus procesos deben estar altamente relacionados entre sí y enfocarse en resolver un problema bien determinado. Por el contrario, una función con baja cohesión intenta resolver un problema más global utilizando instrucciones necesarias entre sí pero no relacionadas.

Una función es altamente cohesiva cuando sus acciones se pueden leer como una oración con un solo verbo. En cuanto se note que se realiza más de una acción, entonces la cohesión es muy baja.

Un claro ejemplo sería una función que permita calcular un promedio de tres notas enteras:

```
static void calcularPromedio () {
        Scanner entrada = new Scanner(System.in);
        System.out.print("Ingresá el primer número: ");
        int n1 = entrada.nextInt(); System.out.print("Ingresá el
        segundo número: "); int n2 = entrada.nextInt();
        System.out.print("Ingresá el tercer número: "); int n3 =
        entrada.nextInt();
        double promedio = (n1 + n2 + n3) / 3.0;
        System.out.println("El promedio es " + promedio);
}
```

La función anterior posee baja cohesión, pues hay partes del código que se encargan de pedirle datos al usuario, otras que realizan el cálculo del promedio y otras que se encargan de mostrar los resultados por consola.

Una función **calcularPromedio()** altamente cohesiva simplemente recibe los números como parámetros, los procesa y devuelve el resultado a quien la invocó:

```
static double calcularPromedio (int n1, int n2, int n3) { return
        (n1 + n2 + n3) / 3.0;

}
```

Las responsabilidades de pedirle los números al usuario y de mostrar el resultado en pantalla deben ser abordadas por otras funciones.

Acoplamiento

El concepto de **acoplamiento** está muy relacionado con la cohesión, y tiene que ver con la interdependencia entre funciones.

Lo que se busca es tener un programa con bajo acoplamiento, es decir, que las funciones componentes sean lo más independientes entre sí. Por el contrario, un programa con alto acoplamiento posee partes muy dependientes de otras. Lograr un bajo acoplamiento de las partes y que las mismas posean alta cohesión, es el ideal de diseño del software.

Continuando con el programa que permite pedir tres números al usuario y calcular su promedio, te muestro un diseño poco conveniente, donde las partes están muy acopladas:

```
1  package prueba;
2
3  import java.util.Scanner;
4  public class Prueba {
5
6          public static void main(String[] args) {
7                  Scanner entrada = new Scanner(System.in);
8                  double promedio;
9                  System.out.print("Ingresá el primer número: ");
10                 double n1 = entrada.nextDouble();
11                 System.out.print("Ingresá el segundo número: ");
12                 double n2 = entrada.nextDouble();
13                 System.out.print("Ingresá el tercer número: ");
14                 double n3 = entrada.nextDouble();
15                 promedio = calcularPromedio( (int) n1, (int) n2, (int) n3);
16                 System.out.println("El promedio es " + promedio);
17         }
18
19         static double calcularPromedio(int a, int b, int c) {
20                 return (a + b + c) / 3.0;
21         }
22 }
```

La función main() está demasiado acoplada con la función calcularPromedio(), pues para que calcularPromedio() funcione bien, main() debe castear los números ingresados en double como int.

Además, main() es muy poco cohesivo: se encarga de pedir los números, castearlos, enviarlos como argumentos a la función calcularPromedio() y luego mostrar el resultado.

Un rediseño del programa anterior agrega más funciones que se encarguen de pequeñas soluciones. La función main() se encarga de unir todas las partes para cumplir con el objetivo:

Es preferible tener muchas función es altamente cohesivas y con poco acoplamiento que unas pocas funciones que realicen multitud de acciones, unas dependientes de otras.

```
1  package prueba;
2
3  import java.util.Scanner;
4  public class Prueba {
5          public static void main(String[] args) {
6                  int n1 = obtenerEnteroDesdeConsola("Ingresá el primer número: ");
7                  int n2 = obtenerEnteroDesdeConsola("Ingresá el segundo número: ");
8                  int n3 = obtenerEnteroDesdeConsola("Ingresá el tercer número: ");
9                  double promedio = calcularPromedio(n1, n2, n3);
10                 System.out.println("El promedio es " + promedio);
11         }
12
13         static double calcularPromedio(int a, int b, int c) {
14                 return (a + b + c) / 3.0;
15         }
16
17         static int obtenerEnteroDesdeConsola (String mensaje) {
18                 Scanner entrada = new Scanner(System.in);
19                 System.out.print(mensaje);
20                 double num = entrada.nextDouble();
21                 return (int) num;
22         }
23 }
```