Capítulo 6

MÉTODOS EN JAVA

1. INTRODUCCIÓN

En este capítulo se estudiará cómo incorpora Java los tipos de subprogramas estudiados en el capítulo anterior, así como, los distintos pasos de parámetros y las demás características del lenguaje respecto a la implementación de subprogramas.

También se tratará una técnica de construcción de subprogramas conocida como **recursividad**, que no todos los lenguajes permiten y que supone una ventaja para la construcción de ciertos algoritmos. Consiste en la posibilidad de que un subprograma pueda llamarse a sí mismo.

Java suministra una amplia librería de clases con funcionalidades que abarcan a todos los ámbitos de programación actualmente. Estas clases junto a las implementadas por el programador, ayudarán al desarrollo de aplicaciones, facilitando la legibilidad y el mantenimiento de los programas y la reutilización de código en futuras aplicaciones.

2. MÉTODOS EN JAVA

El término *método* en Java es equivalente al de subprograma, rutina, subrutina, procedimiento o función en otros lenguajes de programación.

En Java, todos los subprogramas son funciones o métodos de clases porque todo está configurado mediante clases. Incluso el programa principal está representado por una función, *main* dentro de una clase que justamente lleva el nombre del programa. Como son funciones todos devuelven “algo” al módulo que los llama, aunque este algo carezca de valor (*void*). Este es pues el mecanismo para implementar procedimientos.

Un método en Java es un bloque con nombre, que incorpora un código ejecutable con un punto de entrada, representado por el nombre del subprograma y un punto de salida, la llave de cierre del bloque o el *return* en caso de que exista.

El código del subprograma se ubica en memoria y se ejecuta cuando es llamado por el programa principal u otro subprograma, en tiempo de ejecución.

Existen dos puntos a considerar en el uso de subprogramas, funciones o procedimientos, su definición y la llamada.

2.1. Definición de métodos en Java

Definir un método o función significa especificar las instrucciones que se ejecutan cuando se llama a dicha función. En Java la definición de un método suele hacerse antes de la función ***main***.

Tiene dos componentes:

- **Cabecera**: determina el tipo de valor que devuelve la función, su nombre y la lista de parámetros si los hay. Es decir, determina su ***prototipo***, ***signatura*** o ***firma*** (¡OJO!, también se le llama *interfaz*, pero no usaremos este nombre para no confundir con el estudio y el establecimiento de la interfaz de la funcionalidad).

  La signatura del método informa de varias cosas:

  - Si el método es **tipo función o tipo procedimiento**.
  - El **tipo del valor devuelto**, aunque este sea ***void***.
  - El **nombre del método**.
  - Los **parámetros requeridos** (ninguno, uno o varios) y sus tipos.

  La signatura del método no informa de:

  - Los requisitos que deben cumplir los parámetros.
  - Qué ocurre en caso de enviar datos erróneos.
  - Las condiciones que pueda haber tras la ejecución del código de la función o procedimiento.

- **Cuerpo**: El cuerpo está formado por declaraciones, expresiones y otro código necesario para realiza la tarea concreta que hace el método. Se configura como un bloque, por tanto, estará delimitado por llaves.

La sintaxis de definición de un método es la siguiente:

//Cabecera
[Especificador]tipoDevuelto nombreMétodo (declaración-paráms-formales)

{
<declaración variables locales> **//Cuerpo.**
<sentencias ó proposiciones>
}

Especificador. De esto hablaremos más tarde, pero por lo que nos interesa ahora diremos que el especificador o modificador puede ser:

- De acceso (`private, public` o `protected`). Los modificadores de acceso, como su nombre indica, determinan desde qué clases se puede acceder a un determinado elemento.

  El nivel de acceso `public` permite acceder al método desde el exterior de la clase donde está definido, independientemente de que esta pertenezca o no al paquete en que se encuentra el elemento. Lo usaremos de momento para poder acceder a las funciones desde cualquier punto del programa principal. Los demás los estudiaremos más adelante.

  ▪ Otros especificadores, por ejemplo `static`. Heredado de la terminología de C++, el modificador `static` sirve para crear miembros que pertenecen a la clase ya sean atributos o métodos, y no a una instancia de esa clase. Esto implica, entre otras cosas, que no es necesario crear un objeto de la clase para poder acceder a estos atributos y métodos. Este es el motivo por el cual es obligatorio que `main` se declare como `static`.

  Otro uso sería el de crear una colección de métodos y atributos relacionados a los que poder acceder sin necesidad de crear un objeto asociado, que podría no ser conveniente, como es el caso de la clase `Math`. Estas suelen llamarse *clases de utilidad*.

tipoDevuelto Se refiere al tipo del dato que devuelve la función mediante la sentencia *return* y debe coincidir con el declarado en su prototipo.

nombreMétodo. Es un identificador y seguirá las reglas establecidas para su construcción. Suele estar formado por un verbo seguido, opcionalmente, del nombre de un objeto o por el nombre del objeto que calcula. Recordad que debe ser autodocumentado.

Convención Java: Los métodos comienzan por minúscula y si es compuesto el resto de palabras comienza por mayúsculas.

Ejemplos: ingresarDinero, potencia, abonarImpuestos

declaración-paráms-formales. En la cabecera de funciones y procedimientos hay que declarar los parámetros formales que deben utilizarse, por lo que se tendrá en cuenta que:

- Hay que especificar tipo e identificador de cada uno de ellos. Si la lista está vacía, no se escribe nada.
- Los parámetros deben coincidir en número y tipo con los descritos en la declaración. Existe una propiedad implementada en Java y heredada de C++, llamada **elipsis**, que permite que el número de parámetros varíe de unas llamadas a otras. Se estudiará más adelante.
- En caso de no existir parámetros explícitos se dejan los paréntesis vacíos.

En Java, la función nace cuando comienza su cabecera y muere (desaparecen de memoria todos sus datos) cuando la línea de control de flujo llega a la llave de cierre del bloque o bien cuando encuentra una sentencia *return*.

Ejemplo para definir la función *dividir* dos números:

```
public static double dividir(int num,double den)//Cabecera
{
 double div;         //Cuerpo
 div = num / den;
 return (div);
}
```

2.2. Llamada a un método

El segundo contexto en el que aparece el nombre de un método es cuando se la **llama** desde el programa principal o desde cualquier otra función. Una función o un procedimiento son referenciados por su nombre, seguido de un conjunto de parámetros actuales o reales, que el compilador hace corresponder con los parámetros formales.

En funciones y en procedimientos, parámetros formales y reales se corresponden uno a uno en orden y debe haber el mismo número de formales que de reales. Además, cada argumento real debe ser del mismo tipo de datos que su formal correspondiente. Si el tipo de un argumento real no coincide con el correspondiente argumento formal, el sistema los convierte al tipo de este último, siempre que se trate de tipos compatibles. Si no es posible la conversión, el compilador dará los mensajes de error correspondientes.

El formato genérico de llamada a un método es:

nombreMétodo ([lista-parámetros-actuales])

La lista de parámetros actuales son los valores de llamada al método separados por comas.

Generalmente, como resultado de la ejecución del cuerpo de un función, se devuelve un valor asociado a su nombre a través de la sentencia *return* (que se verá a continuación) o, en otros casos, la función se comporta como un procedimiento y no devuelve ningún valor asociado al nombre, siendo el valor de retorno un tipo *void*.

La llamada a una función propiamente dicha, es decir, con valor de retorno, no puede aparecer a la izquierda de un símbolo de asignación. Por esta razón, aparecerá siempre a la derecha de dicho símbolo, para asignar el valor asociado a su nombre a la variable que aparece a la izquierda. También puede formar parte de una expresión compleja o aparecer sin ser asignada a ninguna variable.

Ejemplos:

```
System.out.print("tengo "+edad+" años");//llamada correcta
                                          // a print
```

Siendo `dividir` y `potencia` funciones definidas en el mismo programa que las llama:

```
res = dividir (x, y); // llamada correcta a dividir
potencia(base, expo)=resultado; //Llamada ilegal a Potencia
```

La siguiente es una llamada inusual a *Potencia*, pero no ilegal. Aunque se realiza el proceso correspondiente, no se dispone del valor que devuelve, puesto que no se ha almacenado en ninguna variable. Este tipo de detalle puede observarse evaluando la función con el depurador del entorno integrado en ejecución paso a paso.

Llamadas correctas a distintas funciones y procedimientos

```
System.out.print("La potencia es :"+ potencia(base, expo));
resultado = auxiliar + potencia (2,3);
System.out.print("La potencia es :"+ Math.pow(2.0,3.0));
```

Nota: `out` es un atributo de clase (`static`) de la clase System, por eso se llama así la función `print`.

3. SENTENCIA RETURN

La sentencia *return* se utiliza para salir inmediatamente de un método y para devolver valores. A menos que el valor de retorno sea *void*, el método contendrá una sentencia *return*. Tras la salida de la función se vuelve al programa o subprograma que hizo la llamada y al lugar de llamada de dicha función.

Aunque es cierto que Java permite el uso de varias sentencias *return* en una misma función, no se hará uso de esta característica, puesto que se ha tomado como premisa, por razones de calidad, que los subprogramas tienen, al igual que el programa principal, un único punto de entrada y un único punto de salida.

La sintaxis es la siguiente:

return [(expresión)];

Donde *expresión* representa el valor de retorno de la función, que como puede observarse, es opcional. En caso de existir, es único en cada llamada y debe coincidir con el tipo de dato declarado en la cabecera del método. Por el contrario, si no existe, la función se comportará como un procedimiento y devolverá un *void*.

Cuando el compilador encuentra una sentencia *return*, salta inmediatamente a la llave de cierre de la función y termina su ejecución.

El compilador de Java comprueba que exista una sentencia *return* al final de un método que deba devolver un valor. Si no es así, nos dará el error

```
Missing return statement
```

El compilador también detecta si hay algo después de la sentencia *return* (un error porque la sentencia *return* finaliza la ejecución de un método y nunca se ejecuta lo que haya después):

```
Unreachable statement
```

3.1. Vuelta de un método

Existen dos formas de volver desde un método al programa principal o al subprograma que realiza la llamada:

1. Cuando se ejecuta la última sentencia y se encuentra la llave de cierre del método.

   Ejemplo:

   ```
   void menu ()
   {
      System.out.println ("\nPara sumar pulse S");
      System.out.println ("\nPara restar pulse R");
      System.out.println ("\nPara dividir pulse D");
      System.out.println ("\nPara Multiplicar pulse M");
      System.out.println ("\nPara finalizar pulse F");
    }//Vuelta al programa que realiza la llamada.
   ```

   ¡Ojo!, un procedimiento también puede acabar con una sentencia *return* sin parámetro.

2. A través de la sentencia *return*.

   Java permite que una función pueda tener más de una sentencia *return*, aunque es aconsejable no hacer uso de esta característica por razones de calidad.

Ejemplo:

```
int potencia (int base, int exp)
{
 int result = 1;
 if (exp < 0)
   return -1;
 /*Salida inmediata devolviendo -1 por no poder
 calcular potencias de exponente negativo. Se salta
 a la llave de cierre de la función.*/
 /*No es necesario else ya que si se verifica la
 condición del if se sale inmediatamente */
 for (; exp; exp--)
       result = base * result;
 return (result);/*Cuando lo encuentra salta a la
                  llave de cierre de la función.*/
}
```

Es muy sencillo cambiar el código para evitar el uso de varios *return*:

```
int potencia (int base, int exp)
{
 int result = 1;
 if (exp < 0)
   result = -1;
 else
   for (; exp; exp--)
     result= base * result;
 return (result);
}
```

3.2. Ejemplos sencillos

Seguidamente se muestra un ejemplo de declaración y uso de un método que devuelve el cubo de un valor numérico real con una sentencia *return*:

```
/**
* PruebaDividir.java
*/
public class PruebaDividir
{
   public static double dividir(int num, double den)//Cabecera
   {
    double div;        //Cuerpo
    div = num / den;
    return (div);
   }
   public static void main (String [] args)
   {
```

```
        double res;
        res= dividir(8,2.0);
        System.out.println("El resultado es: "+res); //llamada
    }
}
```

Al igual que en C++ en Java, la definición del método puede realizarse en el código fuente antes o después de main, pero suele hacerse delante. En el caso anterior, `public` y `static` son los modificadores especificados en la cabecera del método. El uso de estos dos modificadores permite que los métodos Java sean similares a las funciones C++.

En este caso, el tipo de dato que debe indicar en la cabecera de declaración del método es el tipo `void` no tiene porqué usarse la sentencia `return`, pero si se usa no viene seguida de ninguna expresión.

En el siguiente código se incluye un ejemplo de método que no devuelve un valor, es decir, se comporta como un procedimiento:

```
/**
* PruebaTabla.java
*/
public class PruebaTabla
{
   public static void main (String [] args)
   {
    calcularTabla(4); // ejemplo de llamada
    calcularTabla(7);
   }
   public static void calcularTabla (int n) // Tipo Procedimiento
   {
    int i, producto;
    System.out.println("Tabla de multiplicar del numero " + n);
    for (i=0; i<=10; i++)
    {
          producto = n*i;
          System.out.println(n + " x " + i + " = " + producto);
    }
    return; // No devuelve ningún valor
   }
}
```

3.3. Ejemplo de ejecución paso a paso

Cuando se invoca un método, el ordenador pasa a ejecutar las sentencias definidas en el cuerpo del método:

```
public  class  Mensajes
```

```
{
   private  static  void  mostrarMensaje  (String  mensaje)
   {
       System.out.println("***  "  +  mensaje  +  "  ***");
   }fin de mostrarMensaje

   public  static  void  main  (String[]  args)
   {
       mostrarMensaje("Hola"); // ...
       mostrarMensaje("Adios");
   }//fin de main
}//fin de la clase
```

Al ejecutar el programa:

1. Comienza la ejecución de la aplicación, con la primera sentencia especificada en el cuerpo del método ***main***.
2. Desde ***main***, se llama a `mostrarMensaje` con, por ejemplo, "Hola" como parámetro. Se pasa a ejecutar el método.
3. El método `mostrarMensaje` muestra el mensaje de saludo decorado y termina su ejecución.
4. Se vuelve al punto donde estábamos en ***main*** y se continúa su ejecución.
5. Justo antes de terminar, volvemos a llamar a `mostrarMensaje` para mostrar un mensaje de despedida.
6. `mostrarMensaje` se vuelve a ejecutar, esta vez con, por ejemplo, "`Adios`" como parámetro. Se muestra el mensaje de despedida.
7. Se termina la ejecución de `mostrarMensaje` y se vuelve al método desde donde se hizo la llamada (***main***).
8. Se termina la ejecución del método ***main*** y finaliza el programa.

4. FUNCIONES Y PROCEDIMIENTOS

Como se vio en el capítulo anterior, funciones y procedimientos son subprogramas con algunas diferencias en el encabezado, la finalización y la llamada. Véase en el siguiente esquema esas diferencias:

	FUNCIONES	PROCEDIMIENTOS
Llamada	VAR = nombre (parámetros reales);	nombre (parámetros reales);
Cabecera	tipo nombre (decl_par_formales)	void nombre (decl_par_formales)
return	Sí	No

Ejemplo: observa las diferencias existentes en la llamada, cabecera y finalización de la función y el procedimiento siguientes.

	FUNCIÓN	PROCEDIMIENTO
Llamada	`max=maximo(aa,bb);`	`error (z);`
Cabecera	`int maximo(int a,int b)`	`void error (int x)`

Cuerpo	`{`
	` int c;`
	` if (a>b)`
	`  c=a;`
	` else`
	`  c=b;`
	` return (c); //Finalización`
	`}`
		`{`
		` switch (x)`
		` {`
		` case 1:`
		`  System.out.print ("Error: incorrecto")`
		`  break;`
		` case 2:`
		`  System.out.print ("Error: inexistente")`
		`  break;`
		` }`
		`} //Finalización`

5. ÁMBITO Y TIEMPO DE VIDA

Respecto al tiempo de vida y ámbito de los objetos que forman parte de un método en Java, se consideran las siguientes características:

- El concepto de ámbito está estrechamente relacionado con el concepto de bloque y es muy importante cuando se trabaja con variables en Java. El ámbito se refiere a cómo las secciones de un programa (bloques) afectan el tiempo de vida de las variables. Ámbito es la parte del programa en que existe y por tanto se conoce y puede accederse un ítem de dato o de sentencia.

- En Java, las funciones se definen dentro de las clases todas al mismo nivel, incluso el programa principal representado por *main*. Su acceso dependerá del modificador de acceso con el que se las defina.

- En Java, el código de un método es privado. Esto significa que el código y los datos definidos dentro, no pueden interactuar con el código y los datos de cualquier otro método. Por tanto, código y datos tienen ámbito local.

- En cuanto a las variables en Java, pueden ser:

    - Variables de instancia: es una variable definida para las instancias de una clase (cada objeto tiene su propia copia de la variable de instancia). El ámbito de una variable de instancia abarca todos los métodos no estáticos de una clase:

        - Cuando es privada, todos los métodos pueden acceder al valor almacenado en la variable de instancia. No accesible desde fuera.

        - Cuando es pública, se puede acceder a ella desde cualquier lugar en el que se disponga de una referencia a un objeto de la clase. Nunca se usarán variables de instancia de este tipo, porque es contrario a la metodología orientada a objetos.

    - Variables de clase: es una variable definida para la clase con el modificador `static` y la variable es compartida entre todas las instancias de una clase. El ámbito de una variable estática:

        - Si es privada, cubre todos los métodos estáticos de la clase en que está definida.

        - Si es pública, abarca todos los métodos estáticos de todas las clases que formen parte de la aplicación.

    - Variables locales: es una variable definida dentro del cuerpo de un método.

        - El ámbito de una variable local comienza en su declaración y termina donde termina el bloque de código ({}) que contiene la declaración.

        - Los parámetros se comportan como variables locales de los métodos.

    - **En Java no existen variables globales, pero, como se ha dicho anteriormente, existen las variables de clase, que funcionan de manera parecida (serán estudiadas en capítulos posteriores)**.

6. ARGUMENTOS DE LOS MÉTODOS

La conexión entre programa y subprograma se realiza por el hecho de compartir la información de las variables que necesita el último, y que posee el primero y viceversa. Esto puede realizarse utilizando objetos de datos (variables, constantes, objetos de clases,…) globales, puesto que son conocidas por programa y subprogramas que este contenga, o mediante el paso de parámetros. Los objetos de datos globales no deben usarse, no sólo por los efectos laterales que pueden provocar, sino, además, por restringir la generalidad del método y por ocupar más memoria, puesto que su tiempo de vida dura mientras dure todo el programa. Por tanto, se utilizará el paso de parámetros.

6. 1. Paso de parámetros en Java (por valor)

El mecanismo de paso de parámetros varía de unos lenguajes a otros. Cuando se realiza una llamada con parámetros a un método, el paso de los argumentos actuales a los formales se hace en Java **solo por valor**.

Tal como se explicó en metodología, en la llamada a una función con paso por valor se produce una copia de los argumentos actuales o reales de la llamada en los formales de la cabecera. El comportamiento es igual que una asignación y se aplica una conversión de tipo automática si procede. Cualquier modificación que se realice dentro de la función sobre los parámetros formales, no afectará en absoluto a los actuales.

¡OJO! Como los nombres de las variables de tipos **no primitivos** son en realidad referencias a objetos (sus direcciones en memoria), lo que se copia en este caso es la referencia del objeto (no el objeto en sí). En realidad este paso de parámetros es igual que el **paso por dirección** de C, dónde lo que se realiza es la copia de una dirección de memoria (que también **refiere** al objeto al igual que su nombre)

Como consecuencia, podemos modificar el estado de un objeto recibido como parámetro si invocamos métodos de ese objeto que modifiquen su estado. La referencia al objeto no cambia, pero sí su estado.

Los tipos primitivos, por tanto, no pueden pasarse por referencia y además no son objetos, por lo que no podemos modificarlos cuando se pasan en la lista de parámetros. Podemos utilizar las clases envolturas, envoltorios o Wrappers que sí son objetos, pero que son inmutables (no se pueden modificar), todos ellos definidos como *public final*. Existen otros objetos inmutables en Java, como por ejemplo las cadenas de caracteres o *String* también definida *public final class String*.

Ejemplo: Realizar un programa para intercambiar dos números si el primero es mayor que el segundo.

```
/*PasoParam.java
*Ejercicio para comprobar el paso de parámetros en Java
```

```
*/
import java.util.Scanner;
import java.io.*;
class PasoParam
{
 /*********************************************************
 Proceso: Esta función intercambia (localmente) el contenido de dos
 variables
 Signatura: public static void intercambiar(double parFor1,double parFor2)
 precondiciones: ninguna
 entradas/salidas: dos números reales
 Nota: Debería tener los dos números intercambiados, pero no puede ser
 porque Java no permite paso por referencia.
 postcondiciones: ninguna
 *********************************************************/
 public static void intercambiar (double parFor1, double parFor2)
 {
  double aux;
  aux = parFor1;
  parFor1 = parFor2;
  parFor2 = aux;
  System.out.println("Dentro de intercambiar:"+parFor1+", "+parFor2);
  }//fin intercambiar

  public static void main(String[] args)throws java.io.IOException
  {
   double numero1, numero2;
   Scanner teclado = new Scanner (System.in);

   System.out.print("Escribe dos número reales:"+'\n'+"Radio:  ");
   numero1 =teclado.nextDouble();
   numero2 =teclado.nextDouble();
   System.out.println("Antes del intercambio:"+numero1+", " +numero2);
   if (numero1>numero2)
   {
      intercambiar (numero1, numero2); //llamada
      System.out.println("Después del intercambio:"+numero1+"," +numero2);
  }
  }//fin main
}//finclase
```

Como se ha dicho anteriormente, en Java los tipos primitivos no son objetos y no son mutables, por tanto, aunque localmente se hace el intercambio en la función intercambiar, el programa principal no se entera del intercambio, puesto que el paso de parámetros se hace por valor.

Se aconseja seguir la traza del programa principal y el subprograma por separado y, después, hacerlo paso a paso conjuntamente. En el primer caso, no se aprecia error, los números son intercambiados tal como pide el enunciado. En el segundo caso, observe que los cambios efectuados en `parFor1` y `parFor2` no se reflejan en los parámetros reales `numero1` y `numero2`. Puede seguir la traza paso a paso con el compilador investigando los valores de todos los parámetros.

7. CÓMO DOCUMENTAR UN MÉTODO

Cuando se realizan librerías de clases se deberá incluir como documentación una ayuda donde venga reflejada la interfaz de cada método de la clase, con el objetivo de facilitar al programador las características de uso.

Cada función deberá incluir al menos:

- Librería donde se encuentra, o árbol jerárquico de clases en su caso.
- Signatura (llamada también prototipo, cabecera o firma).
- Descripción: Breve información sobre uso de la función, qué hace y en qué contexto. Es decir, deben describirse en lenguaje natural:
  - Características de las entradas, salidas, precondiciones y postcondiciones.
  - Relaciones con otras funciones.
- Ejemplo de uso. Programa pequeño de muestra del uso de la función.

Se puede observar como lo hacen algunos lenguajes de programación como Java o C++. Por ejemplo, en Java las funciones *pow* o *sqrt* de la clase **Math**.

8. RESGUARDOS Y CONDUCTORES

Resguardos y conductores son herramientas sencillas que nos ayudan en el proceso de implementación de software.

8.1.Resguardos

Una de muchas ventajas en la construcción de software en base a módulos es la posibilidad de comenzar a probar el diseño antes de haber escrito el código de todos los subprogramas. Por ejemplo, es posible escribir pequeños procedimientos vacíos para los módulos que aún no hayan sido implementados. **Subprogramas vacíos** son aquellos que no contienen ningún código o bien un código no significativo. A estos subprogramas se les conoce como **resguardos** y, generalmente, llevan una sentencia *print* con un mensaje simple, por ejemplo, “Llamada al procedimiento, función o

método tal". De esta forma, aunque el subprograma no realice ningún proceso, permitirá observar si se ha llamado o no en el momento correcto.

El resguardo también puede utilizarse para escribir en pantalla los valores de los parámetros enviados en la llamada, con lo que se observa si el paso se realiza correctamente. Por otro lado, también puede servir para simular el resultado que se debe calcular, es decir, el valor devuelto, por lo que puede ser utilizarlo para controlar las condiciones de ejecución de la prueba y para comprobar si la llamada se hace correctamente.

Un resguardo tiene el mismo nombre e interfaz que el subprograma al que simula y, sea cual sea el diseño que acabe teniendo este último, el resguardo es siempre mucho más sencillo.

Ejemplo: Realizar un resguardo para calcular el máximo común divisor de dos números y otro para el procedimiento cuyo prototipo es: *void pintarCar (int n, char a)*.

En ambos casos lo primero será estudiar las interfaces.

```
/************************************************************
 * Función para calcular el máximo común divisor de dos números
 * Signatura: double maxComunDiv(double num1, double num2);
 * Entradas: dos números reales.
 * Precondiciones: Los números de entrada no pueden ser 0.
 * Salidas: un número real.
 * Postcondiciones: Asociado al nombre de la función se dispone
 * del máximo común divisor de los números de la entrada o 0
 * si alguno de ellos es 0.
 ************************************************************/
```

Acorde con la interfaz diseñada hacemos el resguardo.

```
double maxComunDiv (double num1, double num2)
{
 System.out.println("Llamada al método maxComunDiv:  ");
 return 1;
}
```

Y para la segunda funcionalidad:

```
/************************************************************
Este procedimiento muestra en pantalla n veces un carácter.
signatura: void pintarCar (int,char);
Entradas: un número y un carácter
Precondiciones: El número debe ser entero mayor que 0 y el
carácter imprimible
Salidas: no tiene. Se muestra en pantalla el carácter n veces.
Postcondiciones: ninguna
************************************************************/
```

```
void pintarCar( int n, char a)
{
 System.out.println("Llamada al método pintarCar en
 construcción:  ");
 System.out.println ("\nParámetros:"+ n", "+a);
}
```

En el programa principal que los use, aparecerán las llamadas oportunas a estos resguardos, lo que permitirá realizar el resto de los módulos, incluso el programa completo, a la espera de implementar estos subprogramas más adelante. Además, se podrá observar si las llamadas a los subprogramas se hacen adecuadamente.

8.2. Conductores

Si se tienen problemas para la depuración de un subprograma en particular o bien se están diseñando subprogramas para incluirlos en librerías de usuario, es posible diseñarlos, probarlos y depurarlos utilizando un aislamiento completo. Se utilizará para ello lo que se conoce como un **programa vacío** o programa **conductor**. El conductor es un programa principal que contiene un código mínimo, con las definiciones suficientes para realizar las llamadas a los procedimientos y funciones que se desean verificar y depurar.

Ejemplo: Implementar un subprograma para calcular el máximo común divisor de dos números y otro para pintar en pantalla un carácter las veces que indique un número ambos pasados como parámetros cuyo prototipo es:

```
void pintarCar( int n, char c)
```

Se utilizarán para su implementación, depuración y prueba los siguientes programas conductores.

```
/* Nombre del programa: ConducMax.java
 * Conductor para probar la función maxComunDiv
* */
import java.util.Scanner;
import java.io.*;
import java.lang.*;
public class ConducMax
{
  /************************************************************
   * Función para calcular el máximo común divisor de dos números
   * Signatura: double maxComunDiv(double num1, double num2);
   * Entradas: dos números reales.
   * Precondiciones: Los números de entrada no pueden ser 0.
   * Salidas: un número real.
   * Postcondiciones: Asociado al nombre de la función se dispone
   * del máximo común divisor de los números de la entrada o 0
   * si alguno de ellos es 0.
```

```
    ************************************************************/
      static double maxComunDiv(double num1, double num2)
      {
       int cont;
       double menor, mayor;
        /* Por si alguno de ellos es negativo, se halla el valor
        absoluto de los números*/
       num1 = Math.abs (num1);
       num2 = Math.abs (num2);
       //calcular el menor
       if (num1<num2)
       {
          menor = num1;
          mayor = num2;
       }
       else
       {
          menor = num2;
          mayor = num1;
       }
       for(cont=(int)menor;((int)mayor%cont!=0)||((int)menor%cont!=0);cont--);
       return cont;
      } // fin MaxComunDiv

      public static void main (String[] args)throws java.io.IOException
       {
         double numero1, numero2, maximo;
          Scanner teclado = new Scanner (System.in);

         System.out.print("Escribe dos números :"+'\n'+"Radio:  ");
          numero1 =teclado.nextDouble();
         numero2 =teclado.nextDouble();

         maximo = maxComunDiv(numero1,numero2);
         System.out.print("El MCD de "+numero1+" y de "+numero2+" es:
    "+maximo);
       }
    }
************************************************************
    /* Nombre del programa: ConducPintarCar.java
     * Conductor para probar el procedimiento pintarCar
    * */
    import java.util.Scanner;
    import java.io.*;
```

```
/************************************************************
Este procedimiento muestra en pantalla n veces un carácter.
signatura: void pintarCar (int,char);
Entradas: un número y un carácter
Precondiciones: El número debe ser entero mayor que 0 y el
carácter imprimible
Salidas: no tiene. Se muestra en pantalla el carácter n veces.
Postcondiciones: ninguna
************************************************************/
public static void pintarCar (int n, char c)
{
 int i;
 for(i=0; i<n; i++)
   System.out.print(c);
} // fin PINTAR_CAR

... void main(void)
{
 int num=0;
 char car;
   ..........

 pintarCar (num,car);

} //fin main
```

Los resguardos y conductores se utilizan también cuando los programas son desarrollados por un equipo de programadores, lo que ocurre cuando se acometen proyectos grandes. En estos casos, se realiza en primer lugar el diseño global y las interfaces para la comunicación entre los distintos módulos. A continuación, cada programador diseñará el grupo de módulos que se le asigne, y utilizará conductores y resguardos para implementar, depurar y probar el código. Por último, cuando todos los subprogramas estén codificados y probados suficientemente, se ensamblarán en el proyecto completo.

Para que este método de trabajo funcione, es imprescindible que todas las interfaces hayan sido establecidas explícitamente y que los subprogramas respeten escrupulosamente las especificaciones de dichas interfaces.

Esta metodología de diseño permite la inclusión de los subprogramas en librerías de usuario que podrán ser utilizadas en aplicaciones posteriores.

9. RECURSIVIDAD

A veces puede aparecer la llamada a una función en una expresión dentro del propio código de esa misma función. Una situación en la que una función se llama a sí

misma se conoce con el nombre de **llamada recursiva**. La capacidad de una función o procedimiento para llamarse a sí mismo se conoce como **recursividad**, y a los algoritmos así resueltos **algoritmos recursivos**.

La recursividad es una técnica que puede utilizarse en lugar de la iteración. Las soluciones recursivas son, generalmente, menos eficientes que las soluciones iterativas para el mismo problema. Sin embargo, algunas tareas que tienen soluciones recursivas elegantes y simples, son excesivamente complicadas de resolver iterativamente.

La técnica recursiva para la resolución de algoritmos se encuentra implementada en los lenguajes más modernos, y se usa principalmente para el diseño de sistemas y en el campo de la inteligencia artificial.

REQUERIMIENTOS PARA DISEÑAR UN PROGRAMA RECURSIVO

Para desarrollar un algoritmo recursivo se considerarán los siguientes puntos:

- Debe existir una definición recursiva del proceso. La definición se hará en función de versiones menores de sí mismo, esto se conoce como **caso general** del algoritmo.

  El caso general puede ser único o pueden existir varios.

- Dado que un algoritmo recursivo se expresa en función de una llamada a versiones menores a sí mismo, deberá terminar alguna vez. Es decir, deberá tener un **caso base,** que es aquél para el que la solución se establezca de forma no recursiva, o caso para el que se conoce explícitamente la respuesta. Se soluciona, generalmente, con una sentencia condicional *if* o *switch*.

  El caso base también puede no ser único, independientemente de que existan o no varios casos generales.

  Si en un algoritmo recursivo no se considera el caso base, se producirá una situación llamada de *recursividad infinita*, que produce el mismo efecto que un bucle infinito.

- Seguir la traza de este tipo de algoritmos no es sencillo, por lo que considerando que la llamada recursiva implementada en el lenguaje funciona correctamente, se comprobará tan solo que el algoritmo funciona.

TIPOS DE RECURSIVIDAD

A) Atendiendo a la forma de la llamada existen dos tipos de recursividad:

- **Recursividad indirecta o cruzada**. Es el caso de un subprograma *A* que llama a otro subprograma *B* que llama al primero.

- **Recursividad directa**. Cuando un subprograma se llama a sí mismo.

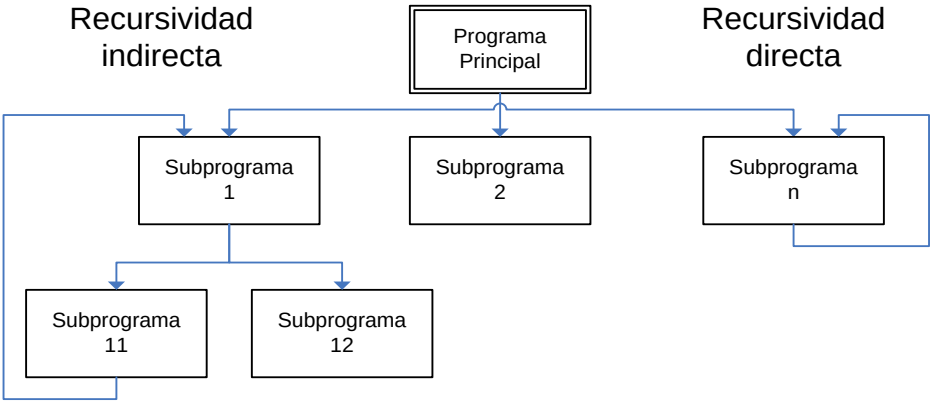

Figura 6.2:Recursividad directa e indirecta

B) Otra forma de clasificar la recursividad

- **Recursividad simple.** Aquella en cuya definición sólo aparece una llamada recursiva. Se puede transformar con facilidad en algoritmos iterativos.

- **Recursividad múltiple.** Se da cuando hay más de una llamada a sí misma dentro del cuerpo de la función, resultando más difícil de hacer de forma iterativa. Un ejemplo típico es la función de Fibonacci.

- **Recursividad anidada**. En algunos de los argumentos de la llamada recursiva hay una nueva llamada a sí misma. Ejemplo típico La función de Ackermann.

¡Cuidado! Tenga en cuenta que en cada llamada deben guardarse en la pila todos los parámetros que habitualmente se almacenan con la llamada a un subprograma, cosa indispensable para hacer posible la vuelta al programa que lo llamó y que, precisamente, en el algoritmo recursivo es él mismo. *Stack overflow* es un error del compilador, que informa que el sistema ha agotado la pila en tiempo de ejecución, por el total consumo del recurso, debido a la cantidad de objetos almacenados en las sucesivas llamadas recursivas. Si esto ocurre, deberá estudiarse una solución distinta para resolver el problema planteado.

EJERCICIO PRÁCTICO DE APLICACIÓN

Realizar una función recursiva para calcular el factorial de un número.

Se exponen las dos soluciones para observar las diferencias.

Interfaz:

Prototipo: `int factorial (int);`

Precondiciones: el número debe ser entero positivo.

Entrada: un número.

Salida: un número resultado de hallar el factorial del número de entrada.

Postcondiciones: Asociado al nombre de la función se devuelve el entero correspondiente al factorial del número de entrada.

Evidentemente la interfaz es independiente del algoritmo que se implemente para solucionar el problema.

Solución iterativa del problema

Definición iterativa de factorial: **fact n = n*(n-1)*(n-2)*. . . *2*1**

```
int factorial1(int num)

{
 int solucion = 1, cont;
 for (cont=1 ;cont <= num; cont++)
     solucion = solucion * cont;
 return  (solucion);
}
```

Solución recursiva

Definición recursiva de factorial: **Caso general: n! = n*(n-1)!**

Caso base: 0! = 1

```
int factorial2 (int num)
{
  int solucion;
  if (num == 0)     // caso base
     solucion=1;
   else             // llamada a un caso más simple
     solucion = num * factorial2 (num - 1);
   return (solucion);
}
```

Traza del algoritmo

La traza del algoritmo iterativo es simple, por lo que se realiza la traza del algoritmo recursivo, suponiendo que se desea calcular el factorial de 3, es decir, se ejecuta la función *factorial2* con *num* = 3.

Llamadas recursivas:

- ¿Cuánto vale el factorial de 3?: 3 por el factorial de 2.
- ¿Cuánto vale el factorial de 2?: 2 por el factorial de 1.
- ¿Cuánto vale el factorial de 1?: 1 por el factorial de 0.
- ¿Cuánto vale el factorial de 0?: 1.

Al llegar al caso base no se realizan más llamadas y comienza la vuelta atrás.

- Factorial de 1 es 1* 1 = 1.
- Factorial de 2 es 2* 1 = 2.
- Factorial de 3 es 3* 2 = 6.

SOLUCION = 6

El gráfico correspondiente al proceso sería el siguiente:

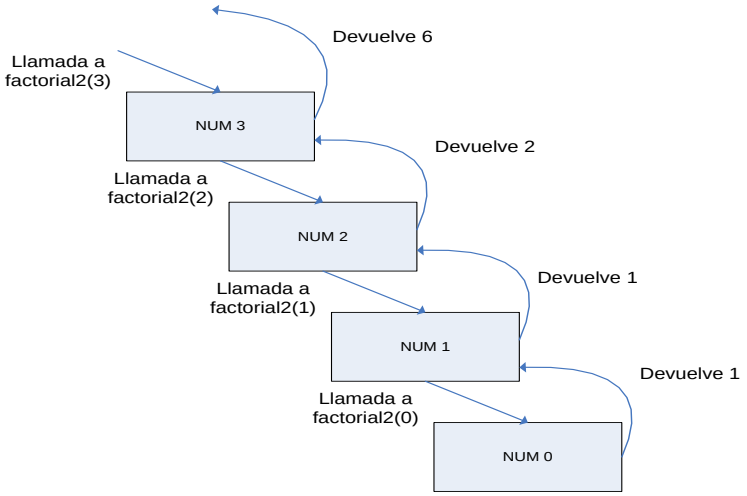

Figura 6.3:Llamadas recursivas

RECURSIVIDAD O ITERACIÓN ¿QUÉ ELEGIR?

Cuando se utiliza una estructura de control de flujo iterativa (*while*, *do while* o *for*), los procesos se ejecutan por repetición del código que contienen, controlando las iteraciones con una condición. Sin embargo, en los algoritmos recursivos ese proceso se hace por repetición de sucesivas llamadas a sí mismo, utilizando una sentencia de bifurcación para controlar el flujo de las llamadas. Cada vez que se hace una llamada a un subprograma, deberá asignarse espacio en la pila para todos los objetos locales, por lo que se consumirán más recursos en la llamada recursiva que en la solución iterativa del problema. Por otro lado, el tiempo de ejecución se incrementará con la solución recursiva, puesto que se consume más tiempo en las llamadas a subprogramas. Por estas razones, si la solución iterativa es sencilla deberá aceptarse como la mejor, ya que resultará más rentable y eficiente. No obstante, existen algoritmos cuya solución iterativa es complicada y la recursiva es sencilla por la propia definición del problema. Por ejemplo, el factorial de un número tiene definición recursiva propia. En estos casos, habrá que considerar positivamente el diseño recursivo del problema. En cualquier caso, será el propio diseñador quien evalúe la conveniencia de una u otra solución en razón de los factores considerados como críticos.