FAMILIA PROFESIONAL:	**Informática y Comunicaciones**
CICLOS FORMATIVOS:	**Desarrollo de Aplicaciones Multiplataforma, Desarrollo de Aplicaciones Web**
MÓDULO:	**Programación**

UNIDAD 2: PROGRAMACIÓN MODULAR

CONTENIDOS

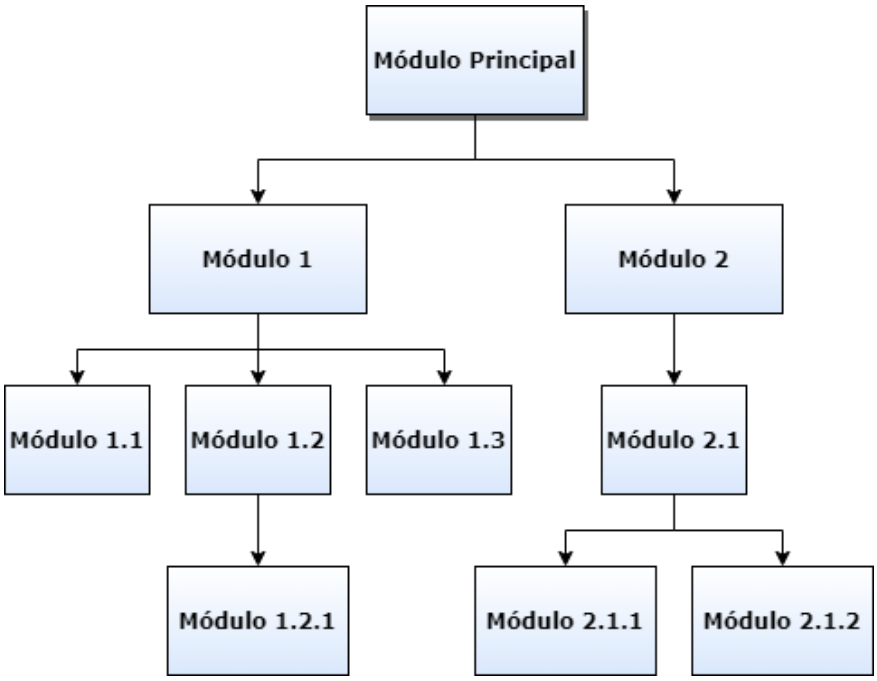
AUTORES: **Fernando Rodríguez Alonso**
Sonia Pasamar Franco

ÍNDICE DE CONTENIDOS

1. DISEÑO DESCENDENTE

El **diseño descendente** (también denominado *top-down*) consiste en dividir un problema dado en diversos subproblemas más pequeños que se pueden resolver por separado y después recomponer los resultados para generar la solución al problema planteado. A su vez, si es necesario, cada subproblema se puede descomponer en otros problemas aún más pequeños para ser resueltos.

Cada subproblema se llama **módulo**, tiene una misión específica y detallada dentro del problema dado y se puede resolver de forma independiente. Cualquier programa no trivial normalmente se descompone en módulos, creando entre sí una estructura modular jerárquica o en árbol que permite resolver el problema dado.



Las **secciones de un módulo** son:

- **Cabecera o Interfaz.** Incluye la información necesaria para definir el subproblema y poder utilizar este módulo después, como el nombre, los datos de entrada que necesita y los datos de salida que devuelve.
- **Cuerpo o Implementación.** Incluye las secuencias de instrucciones o sentencias, organizadas mediante diversas estructuras de control de flujo, que pueden ejecutarse en un ordenador para resolver el subproblema indicado.

Las principales **ventajas de la programación modular** son:

- Facilita la comprensión del problema y su resolución escalonada.
- Aumenta la claridad y la legibilidad del código fuente del programa.
- Permite la resolución del problema por varios programadores a la vez.
- Reduce el tiempo de desarrollo aprovechando módulos previamente codificados.
- Mejora la depuración del programa, pues se pueden probar los módulos aisladamente de forma más sencilla e independiente.
- Facilita un mejor y más rápido mantenimiento del programa, al poder realizar modificaciones o implementaciones de forma más sencilla tomando como base módulos ya desarrollados.

2. TIPOS DE MÓDULOS

Según la **función de un módulo dentro del programa**, el módulo se puede clasificar en:

- **Módulo Principal.** Tiene como objetivo dar una solución al problema planteado y siempre está presente dentro del programa. También se llama programa principal, ya que el flujo de ejecución del programa comienza y finaliza en este módulo. Si el problema es complejo, será recomendable descomponer el módulo principal en varios módulos secundarios.
- **Módulo Secundario.** Tiene como objetivo dar una solución a una parte o componente del problema planteado (módulo principal) o de otra parte (módulo secundario). Dependiendo de la complejidad del problema, puede haber cero, pocos o muchos módulos secundarios dentro del programa.

Según el **posible retorno de un valor por un módulo**, el módulo se puede clasificar en:

- **Función.** Retorna un valor cuando devuelve el control del flujo de ejecución al módulo que lo ha invocado. El valor devuelto al módulo llamador se debe recoger en una expresión. Por ejemplo:

```
resultado = calcularMediaDeTres(5.83, 4.72, 6.91);
```

- **Procedimiento.** No hace un retorno explícito de un valor al finalizar la ejecución del módulo. Por ejemplo:

```
dibujarRectanguloDeAsteriscos(5, 14);
```

Según el **momento en que un módulo ha sido desarrollado**, el módulo se puede clasificar en:

- **Módulo de Biblioteca.** Ha sido desarrollado con anterioridad al momento de realizar un programa, está contenido en un fichero fuente o compilado de una biblioteca y se puede utilizar en cualquier parte del programa.
- **Módulo de Programa.** Se desarrolla explícitamente como una parte o componente del programa como consecuencia de aplicar el diseño descendente al programa.

Según la **situación de un módulo con respecto al módulo que lo invoca**, el módulo se puede clasificar en:

- **Módulo Interno.** Está incluido en el mismo fichero fuente o compilado donde se encuentra el módulo que lo llama.
- **Módulo Externo.** Está incluido en un fichero fuente o compilado distinto del fichero donde se encuentra el módulo que lo llama.

3. COMUNICACIÓN ENTRE MÓDULOS CON PARÁMETROS

La cabecera de un módulo contiene, además de un nombre que lo identifica, una lista de parámetros, que está compuesta por cero o más parámetros. Un **parámetro** (también denominado argumento) es una variable de enlace que se utiliza para comunicar un valor de un tipo de datos en una llamada o invocación entre el módulo llamador y el módulo llamado.

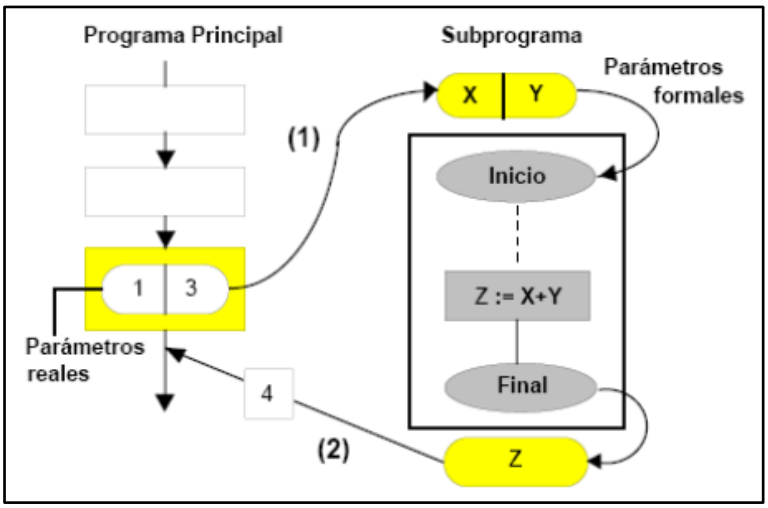
El uso de parámetros aporta dos ventajas:

1) Describe el funcionamiento de un módulo como una **caja negra**, es decir, se conoce los tipos de datos de entrada y de salida del módulo y se ignora cómo trabaja éste internamente.
2) Evita que se produzcan **efectos laterales** en el programa. De esta forma, la modificación de un módulo no afecta o influye en el funcionamiento de otros módulos relacionados.

3.1. TIPOS DE PARÁMETROS

Según el **módulo donde unos parámetros están definidos**, los parámetros se pueden clasificar en:

- **Parámetros Formales.** Son variables definidas en el módulo llamado o invocado. Los valores o referencias de estos parámetros se reciben desde el módulo que realiza la llamada o invocación.
- **Parámetros Actuales o Reales.** Son literales, constantes, variables o expresiones definidas en el módulo que realiza la llamada o invocación. Los valores o referencias de estos parámetros se envían al módulo llamado o invocado.



Debe existir una correspondencia entre los parámetros actuales y los parámetros formales en número, orden y tipos de datos:

- El número de parámetros actuales debe coincidir con el número de parámetros formales.
- Para cada pareja compuesta por un parámetro actual y un parámetro formal que ocupan la misma posición dentro de sus listas respectivas de parámetros, el tipo de datos del parámetro actual debe ser igual o compatible con el tipo de datos del parámetro formal.

Según el **papel de un parámetro dentro del módulo**, el parámetro se puede clasificar en:

- **Parámetro de Entrada.** Indica un valor o referencia de un dato que el módulo necesita para iniciar su ejecución. Se utiliza para suministrar un dato de entrada desde el módulo llamador al módulo llamado.

- **Parámetro de Salida.** Indica un valor o referencia de un dato que el módulo genera y devuelve como resultado. Se utiliza para devolver un dato de salida desde el módulo llamado al módulo llamador.
- **Parámetro de Entrada/Salida.** Indica un valor o referencia de un dato que el módulo necesita para iniciar su ejecución y posteriormente modifica y devuelve como resultado. Se utiliza para suministrar un dato de entrada desde el módulo llamador al módulo llamado y después devolver un dato de salida desde el módulo llamado al módulo llamador.

3.2. PASO DE PARÁMETROS

Existen dos métodos diferentes para el paso de parámetros entre dos módulos al realizar una llamada o invocación: paso por valor y paso por referencia o dirección.

El **paso por valor** permite comunicar datos entre dos módulos en una dirección (del módulo llamador al módulo llamado) y se basa en copiar valores de parámetros actuales a formales:

- En la llamada o invocación al módulo, se copia el valor de cada parámetro actual del módulo llamador en su correspondiente parámetro formal del módulo llamado. Cada parámetro actual es una expresión.
- El módulo llamado no tiene acceso a los parámetros actuales del módulo llamador (y por tanto, no puede modificarlos) y solamente puede modificar los parámetros formales suyos para obtener resultados. No se pueden pasar valores de retorno al punto de llamada, por lo que todos los parámetros del módulo son parámetros de entrada.
- Cuando finaliza la ejecución del módulo, los parámetros y las variables del módulo llamado se destruyen y no se devuelve información al módulo llamador.
- Existe una instrucción de retorno (con una expresión) que permite añadir al módulo un parámetro de salida y que se puede utilizar dentro del módulo llamado para finalizar su ejecución y devolver un valor al módulo llamador. En el retorno de la llamada o invocación al módulo, se devuelve el valor de la expresión indicada en la instrucción de retorno al módulo llamador para que a continuación éste lo asigne a una variable.

El **paso por referencia o dirección** permite comunicar datos entre dos módulos en ambas direcciones y se basa en copiar referencias o direcciones de memoria de parámetros actuales a formales:

- En la llamada o invocación al módulo, se copia la referencia o dirección de memoria de cada parámetro actual del módulo llamador en su correspondiente parámetro formal del módulo llamado. Cada parámetro actual debe ser obligatoriamente una variable.
- El módulo llamador comparte los datos de memoria referenciados por sus parámetros actuales con el módulo llamado. De esta manera, el módulo llamado puede obtenerlos y modificarlos a través de sus parámetros formales. Todos los parámetros del módulo son parámetros de entrada/salida.
- Cuando finaliza la ejecución del módulo, los parámetros y la variables del módulo llamado se destruyen, no se devuelve información al módulo llamador, pero las modificaciones realizadas en las zonas de memoria referenciadas se mantienen y serán visibles para el módulo llamador.

Java utiliza el paso de parámetros por valor, de forma independiente de los tipos de datos de las variables a comunicar entre dos módulos. Cuando un módulo llama a otro módulo:

1) Se copian los valores de los parámetros actuales del módulo llamador en una zona de memoria (pila) dedicada para el programa.
2) Dentro del módulo llamado, los valores copiados en la pila representan los parámetros formales y se interpretan según la complejidad de cada tipo de datos:
   - Si **el tipo de datos de un parámetro formal es un tipo primitivo** (`boolean`, `byte`, `short`, `int`, `long`, `float` o `double`), el módulo invocado recibe una copia del parámetro actual del módulo llamador.

     Por tanto, el módulo invocado no tiene acceso a dicho parámetro actual y no puede modificarlo.
   - Si **el tipo de datos de un parámetro formal no es un tipo primitivo**, se trata de un tipo estructurado que se gestiona en memoria mediante referencias. El módulo invocado recibe una copia del parámetro actual del módulo llamador, siendo esta copia en realidad una referencia o dirección de memoria.

     De esta manera, el módulo llamador comparte los datos de memoria referenciados por dicho parámetro actual con el módulo llamado y hay dos referencias que apuntan a los mismos datos en memoria: la del parámetro actual y la del parámetro formal. En este caso, el módulo invocado sí puede acceder a los datos de memoria referenciados y modificarlos.

4. SUBPROGRAMAS

Cuando un módulo necesita utilizar otro módulo para resolver un problema, debe realizar una **llamada** o **invocación** a dicho módulo. Una llamada o invocación a un módulo se divide en tres pasos:

- El módulo llamador comunica información al módulo llamado a través de una lista de parámetros (definida en su cabecera). El flujo de ejecución del programa se transfiere desde la instrucción de llamada del módulo llamador al inicio del módulo llamado.
- Se ejecutan las instrucciones correspondientes a la implementación del módulo llamado.
- Cuando finaliza la ejecución del módulo llamado, el módulo llamado devuelve resultados al módulo llamador. El flujo de ejecución del programa se transfiere desde el final del módulo llamado a la instrucción siguiente a la de llamada del módulo llamador.

4.1. FUNCIONES

Una **función** es un subprograma que recibe cero, uno o varios parámetros de entrada y produce un valor de salida o resultado (a veces, indicado como parámetro de salida) a partir de dichos parámetros.

La **cabecera, declaración o definición de una función** contiene los siguientes elementos:

- Un nombre o identificador válido asociado con la función.
- Una lista de parámetros formales, en la que cada parámetro formal será una variable únicamente visible dentro de la función, sólo de entrada y definida mediante un identificador y un tipo de datos.
- El tipo de datos correspondiente al valor de retorno de la función.

El **cuerpo de una función** está compuesto por las instrucciones y sentencias de control que son ejecutables en un ordenador para resolver el subproblema indicado. Estas instrucciones deben contener al menos una instrucción *devolver* entre ellas (normalmente, se usa al final del cuerpo). La instrucción *devolver* finaliza la ejecución de la función y devuelve un valor resultado al módulo que llama a la función.

La **declaración y utilización de una función en Java** se muestra en el siguiente ejemplo:

FUNCIÓN	`public static`
	`double calcularMediaDeTres(double num1, double num2, double num3) {`
	`    double media = (num1 + num2 + num3) / 3.0;`
	`    return media;`
	`}`

PROGRAMA PRINCIPAL	`public static void main(String[] args) {`
	`    Scanner teclado = new Scanner(System.in);`
	`    double numA, numB, numC, resultado;`
	`    System.out.print("Introduce el número A: ");`
	`    numA = teclado.nextDouble();`
	`    System.out.print("Introduce el número B: ");`
	`    numB = teclado.nextDouble();`
	`    System.out.print("Introduce el número C: ");`
	`    numC = teclado.nextDouble();`
	`    resultado = calcularMediaDeTres(numA, numB, numC);`
	`    System.out.println("Media de Tres = " + resultado);`
	`}`

4.2. PROCEDIMIENTOS

Un **procedimiento** es un subprograma que recibe cero, uno o varios parámetros de entrada/salida y realiza efectos laterales sobre el programa (modificación de variables compartidas por varios módulos o impresión de textos en consola) a partir de dichos parámetros.

La **cabecera, declaración o definición de un procedimiento** contiene los siguientes elementos:

- Un nombre o identificador válido asociado con el procedimiento.
- Una lista de parámetros formales, en la que cada parámetro formal será una variable compartida por el módulo que lo llama y el procedimiento, de entrada/salida y definida mediante un identificador y un tipo de datos.

El **cuerpo de un procedimiento** está compuesto por las instrucciones y sentencias de control que son ejecutables en un ordenador para resolver el subproblema indicado.

La **declaración y utilización de un procedimiento en Java** se muestra en el siguiente ejemplo:

PROCEDIMIENTO	`public static`
	`void` **`dibujarRectanguloDeAsteriscos`**`(int altura, int anchura) {`
	`    int i, j;`
	`    for (i = 1 ; i <= altura ; i++) {`
	`        for (j = 1 ; j <= anchura ; j++) {`
	`            System.out.print('*');`
	`        }`
	`        System.out.println();`
	`    }`
	`}`

PROGRAMA PRINCIPAL	`public static void` **`main`**`(String[] args) {`
	`    Scanner teclado = new Scanner(System.in);`
	`    int alto, ancho;`
	`    System.out.print("Introduce el alto: ");`
	`    alto = teclado.nextInt();`
	`    System.out.print("Introduce el ancho: ");`
	`    ancho = teclado.nextInt();`
	`    System.out.println("Rectángulo de Asteriscos");`
	`    dibujarRectanguloDeAsteriscos(alto, ancho);`
	`}`

5. ÁMBITO DE IDENTIFICADORES

En un programa que contiene funciones y/o procedimientos, existen varios puntos en los que se pueden declarar identificadores: en la parte de declaraciones del programa principal, en los parámetros formales de cada función y/o procedimiento, y en la parte de declaraciones de cada función y/o procedimiento.

Se denomina **ámbito de un identificador** a la parte, porción o bloque de código fuente en la que dicho identificador es visible y se puede utilizar. Como regla general, el ámbito de un identificador corresponde al programa principal o subprograma en el que dicho identificador se declara o define.

Los identificadores utilizados en programas principales y subprogramas se clasifican en dos tipos: identificadores locales e identificadores globales.

5.1. IDENTIFICADORES LOCALES

Un **identificador local** es aquel que se declara o define dentro de un subprograma (función o procedimiento), de forma que sólo es accesible y se puede utilizar en dicho subprograma. Fuera del subprograma, el identificador local no existe.

Si otro subprograma utiliza otro identificador local con el mismo nombre, ese identificador local se refiere a otra posición de memoria que contiene otro dato para dicho subprograma. Es decir, puede haber varios subprogramas que tengan un identificador local con el mismo nombre.

Un identificador declarado en un bloque es accesible desde ese bloque (se considera local al bloque) y desde los bloques incluidos o anidados dentro del mismo. Si se declaran identificadores con el mismo nombre en un bloque y en otro bloque incluido o anidado en el anterior, el identificador del bloque interno oculta al identificador del bloque externo.

La utilización de identificadores locales tiene varias **ventajas**:

- Hace que los subprogramas sean independientes, debido a una comunicación de datos entre el programa principal y los subprogramas manipulada a través de listas de parámetros.
- Permite dividir grandes proyectos en módulos más pequeños e independientes. Si hay varios programadores en un proyecto, pueden trabajar por separado de forma independiente.

El siguiente ejemplo muestra el **ámbito de identificadores locales en Java**:

PROGRAMA PRINCIPAL Y SUBPROGRAMA	IDENTIFICADOR	ÁMBITO

`public static int calcular(int a) {`
`  int b;`
`  if (...) {`
`  int c; ...`
`  }`
`  ...`
`}`
 	
`public static void main(String[] args) {`
`  int x;`
`  while (...) {`
`  int y; ...`
`  }`
`  ...`
`}`
	`a`	Subprograma `calcular`

	`b`	Subprograma `calcular`
	`c`	Bloque `if` del Subprograma `calcular`
	`args`	Programa Principal `main`
	`x`	Programa Principal `main`
	`y`	Bloque `while` del Programa Principal `main`

5.2. IDENTIFICADORES GLOBALES

Un **identificador global** es aquel que se declara o define en el programa principal y que se puede utilizar en dicho programa principal y en todos sus subprogramas (funciones y procedimientos).

Un identificador declarado fuera de cualquier bloque se considera global y puede ser utilizados desde cualquier punto del programa principal o de cada subprograma.

La utilización de identificadores globales tiene una **ventaja**:

- Permite compartir datos entre el programa principal y sus subprogramas (funciones y procedimientos) o entre varios subprogramas de forma fácil sin necesidad de añadir una correspondiente variable de entrada/salida en las listas de parámetros.

Sin embargo, se desaconseja el uso de identificadores globales debido al **inconveniente**:

- Reduce la independencia entre los distintos subprogramas y el programa principal, y por tanto, hace la programación modular más difícil. Esto se debe a que producen efectos laterales.

El siguiente ejemplo muestra el **ámbito de identificadores globales en Java**:

PROGRAMA PRINCIPAL Y SUBPROGRAMAS	IDENTIFICADOR	ÁMBITO

`public static int x;`
`public static int y;`
 	
`public static int leer() {`
`  int a;`
`  ...`
`}`
 	
`public static void escribir(int n) {`
`  int b;`
`  ...`
`}`
 	
`public static void main(String[] args) {`
`  int c;`
`  ...`
`}`
	`x`	Subprograma `leer`, Subprograma `escribir` y Programa Principal `main`

	`y`	Subprograma `leer`, Subprograma `escribir` y Programa Principal `main`
	`a`	Subprograma `leer`
	`n`	Subprograma `escribir`
	`b`	Subprograma `escribir`
	`args`	Programa Principal `main`
	`c`	Programa Principal `main`

6. RECURSIVIDAD

Se habla de **recursividad** cuando un subprograma (función o procedimiento) se llama o se invoca a sí mismo. La recursividad tiene exactamente la misma potencia expresiva que la iteración, en el sentido de que permite describir los mismos algoritmos o procesos iterativos de forma recursiva. Su uso genera programas recursivos más compactos que sus correspondientes versiones iterativas.

Los subprogramas recursivos son especialmente apropiados si el problema, el cálculo a realizar o la estructura del problema aceptan una **definición recursiva**: el problema tiene casos triviales o sencillos y define casos complejos mediante recursividad. Sin embargo, la existencia de tales definiciones no garantiza que la mejor forma de resolver un problema sea realizar un subprograma recursivo.

Cuando se desarrolla un subprograma recursivo, se debe asegurar de que el número de llamadas sea finito y pequeño. De esta forma, la recursión siempre finaliza (no hay recursión infinita) y la memoria dedicada al programa (pila) tiene suficiente espacio para almacenar, en cada invocación, los parámetros formales, las variables locales y el estado del proceso en curso para recuperarlo después.

La recursividad requiere dos condiciones para su correcto funcionamiento:

- Debe existir al menos una condición o criterio (denominado caso base o trivial) que no requiera una invocación recursiva y que permita finalizar la cadena de llamadas o recursividad generada.
- Las sucesivas invocaciones deben ser efectuadas con versiones cada vez reducidas del problema inicial, de forma que se encuentren más cerca de un caso base o trivial.

6.1. RECURSIVIDAD SIMPLE Y MÚLTIPLE

Cuando un subprograma genera en tiempo de ejecución como máximo una invocación recursiva, dicho subprograma tiene una **recursividad simple o lineal**. Puede ocurrir que un subprograma incluya varias llamadas recursivas dentro de una estructura alternativa, de forma que cada alternativa sólo tenga una llamada recursiva como máximo.

El siguiente ejemplo muestra una **recursividad simple o lineal en Java**:

FUNCIÓN FACTORIAL

```
public static int factorial(int numero) {
    if (numero == 0 || numero == 1) {
        return 1;
    }
    else {
        return numero * factorial(numero - 1);
    }
}
```

Cuando un subprograma genera en tiempo de ejecución dos o más invocaciones recursivas, dicho subprograma tiene una **recursividad múltiple o no lineal**.

El siguiente ejemplo muestra una **recursividad múltiple o no lineal en Java**:

FUNCIÓN FIBONACCI

```
public static int fibonacci(int numero) {
    if (numero == 0 || numero == 1) {
        return numero;
    }
    else {
        return fibonacci(numero - 1) + fibonacci(numero - 2);
    }
}
```

6.2. RECURSIVIDAD DIRECTA E INDIRECTA

Cuando un subprograma contiene una llamada a si mismo, dicho subprograma tiene una **recursividad directa**.

Cuando un subprograma no se llama a si mismo, pero llama a otro subprograma que contiene una llamada directa o indirecta al subprograma, dicho subprograma tiene una **recursividad indirecta**. Por ejemplo:

- Un módulo A llama a un módulo B y el módulo B llama al A. Entonces el módulo A tiene una recursividad indirecta.
- Un módulo A llama a un módulo B, el módulo B llama a un módulo C y el módulo C llama al módulo A. Entonces se forma una cadena de módulos y el módulo A tiene una recursividad indirecta.

Los siguientes ejemplos muestran una **recursividad indirecta en Java**:

RECURSIVIDAD INDIRECTA SIMPLE

```
public static void moduloA(...) {
    ...
    moduloB(...);
    ...
}
public static void moduloB(...) {
    ...
    moduloA(...);
    ...
}
public static void main(String[] args) {
    ...
    moduloA(...);
    ...
}
```

RECURSIVIDAD INDIRECTA ENCADENADA

```
public static void moduloA(...) {
    ...
    moduloB(...);
    ...
}
public static void moduloB(...) {
    ...
    moduloC(...);
    ...
}
public static void moduloC(...) {
    ...
    moduloA(...);
    ...
}
public static void main(String[] args) {
    ...
    moduloA(...);
    ...
}
```

7. BIBLIOTECAS

Una **biblioteca** (o librería) es un conjunto probado, documentado y, a veces previamente compilado, de funciones y procedimientos que se pueden invocar desde otro programa.

Una biblioteca básica debe proporcionar una colección de constantes, estructuras de datos, funciones y procedimientos independientes del tipo de aplicación donde se vaya a utilizar. Esta colección debe ser suficiente para cubrir las necesidades de la mayoría de las aplicaciones en los lenguajes que permitan su uso.

7.1. CARACTERÍSTICAS DE UNA BIBLIOTECA

Las **características de una biblioteca** ideal son:

- **Completa.** La biblioteca debe proporcionar una familia de subprogramas, unidos por una interfaz compartida pero empleando cada representación diferente, de manera que el programador pueda seleccionar las que sean más apropiadas para la aplicación.
- **Adaptable.** Todos los aspectos específicos de la plataforma deben estar claramente identificados y aislados, de manera que puedan realizarse sustituciones y adaptaciones locales (por ejemplo, mediante el uso de funciones propias que intermedien entre el código de la aplicación y las invocaciones a los procedimientos de biblioteca).
- **Eficiente.** Los componentes deben ser de fácil incorporación al código propio (eficiencia en términos de recursos de compilación), utilizar cantidades razonables de memoria y tiempo de ejecución (eficiencia en ejecución) y de uso comprensible y seguro (eficiencia en términos de recursos de desarrollo).
- **Segura.** Es fundamental que la biblioteca esté completamente probada en todos los entornos previsibles. Uno de los indicadores de esa robustez es el uso de excepciones para identificar condiciones para las cuales se violan las precondiciones de un algoritmo. Cuando estas excepciones se generen el sistema debe ser capaz de mantener la estabilidad sin que se produzcan reacciones anómalas, rupturas bruscas de la secuencia de ejecución o corrupciones en el espacio de direcciones del programa.
- **Simple.** Se trata de dotar a la biblioteca de una organización clara y consistente que facilite la identificación y selección de las estructuras y procedimientos adecuados para el fin requerido. Las técnicas de orientación a objetos ayudan a lograr esto.
- **Extensible.** Los desarrolladores propios deben ser capaces de añadir funcionalidad a la biblioteca sin alterar su integridad arquitectónica original.
- **Independiente de la plataforma.** Consiste en hacer la biblioteca lo más independiente posible del hardware y sistema operativo donde finalmente se ejecute la aplicación que se está desarrollando. Para ello se crean bibliotecas abstractas que actúan como interfaz. Estas bibliotecas se conectan de forma transparente para el desarrollador con otras que sí dependen de los servicios de la plataforma. Dicha conexión se puede producir, bien al compilar el código, con lo que habrá que recompilar para cada tipo de plataforma, o bien en tiempo de ejecución de forma dinámica (como ocurre con las bibliotecas de clases de Java).

7.2. TIPOS DE BIBLIOTECAS

Existen varios criterios de clasificación de bibliotecas: según el desarrollador y según la forma de distribución.

7.2.1. Tipos de Bibliotecas según el Desarrollador

Dependiendo del **desarrollador de una biblioteca**, la biblioteca se puede clasificar en:

- **Biblioteca Estándar del Lenguaje de Programación.** Libera al programador de tareas de codificación simples, repetitivas o de bajo nivel.

  Las bibliotecas facilitan la portabilidad de software cuando diferentes sistemas utilizan un mismo estándar de codificación. Para realizar tareas de bajo nivel (entrada y salida de datos, utilización de comunicaciones, etc.), el programador, además de conocer el hardware, necesitaría escribir un código complicado y largo.

  Para evitar esto los fabricantes de compiladores suministran las funciones y procedimientos necesarios para ejecutar estas tareas. Todo este código suele ir suministrado en bibliotecas, con lo que el programador solo necesita conocer la biblioteca y el nombre de la función o procedimiento correspondiente a la tarea a ejecutar. Estas bibliotecas se denominan estándar y las funciones y procedimientos que contienen se llaman predefinidas.

  Todos los lenguajes de programación de alto nivel incluyen bibliotecas estándares para matemáticas, contenedores o colecciones de datos, entrada y salida de datos, interfaces gráficas de usuario, bases de datos y redes e internet.

- **Biblioteca Definida por el Programador** (o biblioteca de usuario). Facilita la reutilización de código para otros proyectos software.

  Cuando un programador necesita utilizar una serie de funciones y procedimientos en distintos programas, no incluirá todos ellos en cada programa. En su lugar, creará un fichero externo al programa en el que incluirá estas funciones y procedimientos.

  Cuando en uno de los programas necesite utilizar alguna de las funciones y procedimientos, sólo tendrá que realizar una llamada a la misma. Además, tendrá que indicar en el programa las bibliotecas a las que vaya a hacer referencia. Estas bibliotecas creadas con este fin por el programador se denominan bibliotecas de usuario.

7.2.2. Tipos de Bibliotecas según la Forma de Distribución

Dependiendo de la **forma de distribución de una biblioteca**, la biblioteca se puede clasificar en:

- **Biblioteca en Código Fuente.** Se añade a un programa en forma de código fuente.

  Al compilar el programa, el fichero objeto resultante de la compilación contendrá el código máquina correspondiente de todos los módulos de la biblioteca. Esto puede ser un inconveniente, pues no siempre se utilizan en un programa la totalidad del código de la biblioteca.

- **Biblioteca Compilada.** Es un fichero objeto generado a partir del código fuente de las funciones y procedimientos de una biblioteca.

  Cuando se usa una biblioteca compilada en un programa, el enlazador (*linker*) extrae sólo los módulos utilizados de esa biblioteca y los añade al programa ejecutable. Tras obtener este programa ejecutable, no es necesaria la presencia de la biblioteca para que el ejecutable funcione.

El inconveniente es que se genera un programa ejecutable de gran tamaño, lo que puede suponer una ocupación excesiva de la memoria RAM del ordenador, pudiéndose llegar incluso al desbordamiento de la memoria. Este sistema se utiliza en programas que no han de compartir recursos con otras aplicaciones.

- **Biblioteca de Enlace Dinámico.** Es un fichero con código ejecutable que mantiene su independencia física del programa principal.

  Cuando se usa una biblioteca de enlace dinámico en un programa, es necesaria su presencia para que el programa funcione adecuadamente.

  La utilidad principal es evitar programas ejecutables de tamaño excesivo, lo que podría provocar un desbordamiento de la memoria del ordenador. Cuando un programa realiza una llamada a un módulo de la biblioteca, solo se carga en memoria el código del módulo llamado y se libera el espacio ocupado por éste cuando se retorna al programa que lo llamó. Este sistema también es ventajoso cuando muchas aplicaciones utilizan de forma común un amplio grupo de módulos: el código de un módulo dado sólo aparece una vez en el soporte físico. El inconveniente principal es la pérdida de velocidad en la ejecución.

  Un ejemplo de este tipo es el archivo DLL (*Dynamic Link Library*) de los sistemas operativos de Windows, aunque en la actualidad la mayoría de los sistemas operativos poseen bibliotecas de enlace dinámico.

8. CASOS PRÁCTICOS

8.1. FUNCIÓN: MEDIA DE TRES NÚMEROS

El siguiente ejemplo muestra la creación y uso de una función. La función *calcularMedia* recibe como parámetros de entrada tres números reales y devuelve como parámetro de salida la media de los mismos.

En el programa principal, se solicita al usuario que introduzca tres números y se llama a la función *calcularMedia* pasándole estos datos. El dato devuelto por la función es asignado a la variable media. Finalmente, se muestra el resultado por pantalla.

```
import java.util.Scanner;

public class Media {

    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);
        double numeroA, numeroB, numeroC;
        double media;
        System.out.print("Introduce el número A: ");
        numeroA = teclado.nextDouble();
        System.out.print("Introduce el número B: ");
        numeroB = teclado.nextDouble();
        System.out.print("Introduce el número C: ");
        numeroC = teclado.nextDouble();
        media = calcularMedia(numeroA, numeroB, numeroC);
        System.out.println("La media de los tres números es " + media);
    }

    private static double calcularMedia(double numA, double numB, double numC) {
        double resultado;
        resultado = (numA + numB + numC) / 3;
        return resultado;
    }

}
```

Un primer ejemplo de ejecución podría ser el siguiente:

```
Introduce el número A: 5,25
Introduce el número B: 4,25
Introduce el número C: 6,25
La media de los tres números es 5.25
```

Un segundo ejemplo de ejecución podría ser el siguiente:

```
Introduce el número A: 4
Introduce el número B: 5
Introduce el número C: 9
La media de los tres números es 6.0
```

Un tercer ejemplo de ejecución podría ser el siguiente:

```
Introduce el número A: 25,35
Introduce el número B: 15,24
Introduce el número C: 10,5
La media de los tres números es 17.03
```

8.2. PROCEDIMIENTO: DIBUJAR CUADRADO

El siguiente ejemplo muestra la creación y uso de un procedimiento. El procedimiento *dibujarCuadrado* recibe como parámetro de entrada un número entero que representa el lado del cuadrado y muestra en pantalla un cuadrado de asteriscos con el lado indicado.

En el programa principal, se solicita al usuario que escriba el lado del cuadrado y se llama al procedimiento *dibujarCuadrado* pasándole este dato.

```
import java.util.Scanner;

public class Cuadrado {

    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);
        int lado;
        System.out.print("Introduce el lado del cuadrado: ");
        lado = teclado.nextInt();
        dibujarCuadrado(lado);
    }

    private static void dibujarCuadrado(int lado) {
        for(int i = 0; i < lado; i++) {
            for(int j = 0; j < lado; j++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }

}
```

Un ejemplo de ejecución podría ser el siguiente:

```
Introduce el lado del cuadrado: 5
*****
*****
*****
*****
*****
```

Otro ejemplo de ejecución podría ser el siguiente:

```
Introduce el lado del cuadrado: 7
*******
*******
*******
*******
*******
*******
*******
```

8.3. ÁMBITO DE VARIABLES

El siguiente ejemplo muestra la visibilidad de diferentes variables desde diferentes bloques. Para lo que se crea un procedimiento *modificarDatos*, que recibe dos parámetros con datos enteros. Este procedimiento asocia valores a sus dos variables locales y a la variable global.

En primer lugar, destacar que se ha creado una variable global, que está fuera del bloque principal y fuera del procedimiento *modificarDatos*. Esta variable será visible y modificable desde todos los bloques.

En el bloque principal, se declaran dos variables locales y se inicializan. A continuación, se muestra su valor, así como el valor de la variable global. Posteriormente, se realiza una llamada al procedimiento *modificarDatos* pasándole el valor de las dos variables locales. Finalmente, se muestra de nuevo el valor de las variables locales y de la variable global.

```
public class AmbitoVaribles {

    public static int variableGlobal = 100;

    public static void main(String[] args) {
        int variableLocal1;
        int variableLocal2;
        variableLocal1 = 3;
        variableLocal2 = 7;
        System.out.println("La variableLocal1 vale " + variableLocal1);
        System.out.println("La variableLocal2 vale " + variableLocal2);
        System.out.println("La variableGlobal vale " + variableGlobal);
        modificarDatos(variableLocal1, variableLocal2);
        System.out.println("TRAS LLAMAR AL PROCEDIMIENTO DE MODIFICAR:");
        System.out.println("La variableLocal1 vale " + variableLocal1);
        System.out.println("La variableLocal2 vale " + variableLocal2);
        System.out.println("La variableGlobal vale " + variableGlobal);
    }

    public static void modificarDatos(int variableLocal1, int variable2) {
        variableLocal1 = 9;
        variable2 = 33;
        variableGlobal = 95;
    }

}
```

Un ejemplo de ejecución es el siguiente:

```
La variableLocal1 vale 3
La variableLocal2 vale 7
La variableGlobal vale 100
TRAS LLAMAR AL PROCEDIMIENTO DE MODIFICAR:
La variableLocal1 vale 3
La variableLocal2 vale 7
La variableGlobal vale 95
```

Como puede observarse en la ejecución del ejemplo, las variables locales no han cambiado su valor. Esto se debe a que las variables son visibles a nivel de bloque y lo que se pasa al subprograma es una copia de las mismas. Los subprogramas tienen sus propias variables locales, que son independientes de las variables locales de otros bloques, aunque tengan el mismo nombre.

8.4. FUNCIONES MEDIA DE TRES NÚMEROS Y LEER DATOS

El siguiente ejemplo muestra la creación y uso de dos funciones. La función *calcularMedia* recibe como parámetros de entrada tres números reales y devuelve como parámetro de salida la media de los mismos. La función *leerDouble* pide al usuario que escriba un número y lo devuelve como parámetro de salida.

En el programa principal, se llama a la función *leerDouble* tres veces y los datos que devuelve se asignan a tres variables. A continuación, se llama a la función *calcularMedia* pasándole estos datos. El dato devuelto por la función es asignado a la variable media. Finalmente, se muestra el resultado por pantalla.

```
import java.util.Scanner;

public class Media3 {
    public static void main(String[] args) {

        double numeroA, numeroB, numeroC;
        double media;
        numeroA = leerDouble();
        numeroB = leerDouble();
        numeroC = leerDouble();
        media = calcularMedia(numeroA, numeroB, numeroC);
        System.out.println("La media de los tres números es " + media);
    }

    private static double leerDouble() {
        Scanner teclado = new Scanner(System.in);
        double numero;
        System.out.print("Introduce el número: ");
        numero = teclado.nextDouble();
        return numero;
    }

    private static double calcularMedia(double numA, double numB, double numC) {
        double resultado;
        resultado = (numA + numB + numC) / 3;
        return resultado;
    }
}
```

Un ejemplo de ejecución podría ser el siguiente:

```
Introduce el número A: 5,25
Introduce el número B: 4,25
Introduce el número C: 6,25
La media de los tres números es 5.25
```

Otro ejemplo de ejecución podría ser el siguiente:

```
Introduce el número A: 25,35
Introduce el número B: 15,24
Introduce el número C: 10,5
La media de los tres números es 17.03
```