La Sénia
Institut d'Educació Secundària
Paiporta

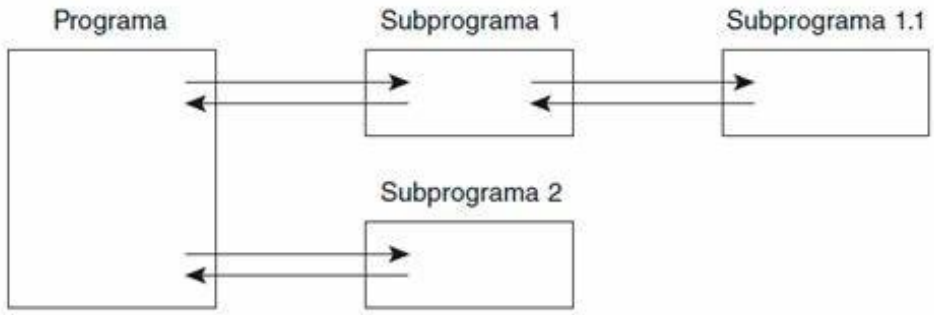
Unión Europea
Fondo Social Europeo
El FSE invierte en tu futuro

Programación Modular

Curso 2023 - 2024

1. Introducción

La programación Modular es un modelo de programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable.



Es una evolución de la programación estructurada para problemas de programación más grandes y complejos.

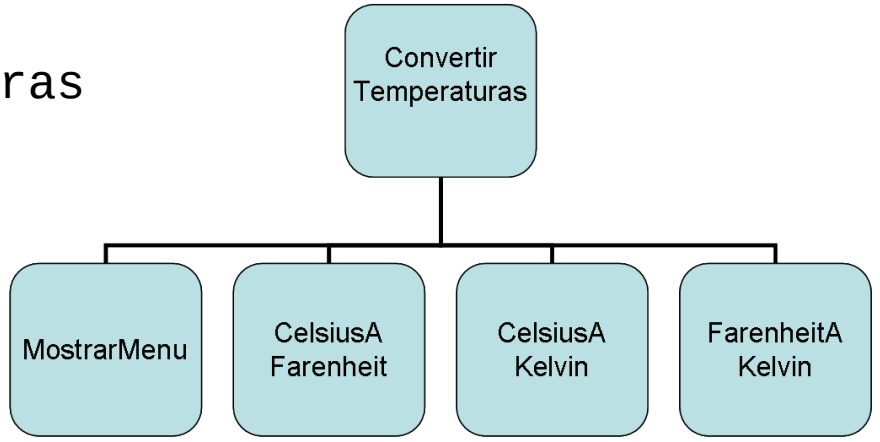
1. Introducción

Un problema complejo es dividido en varios subproblemas más simples.

Los subproblemas pueden dividirse en otros hasta obtener subproblemas lo suficientemente sencillos.

Técnica “Divide y vencerás”

```
Programa para Convertir Temperaturas
Podemos dividirlo en varios
problemas más sencillos como la
conversión a una temperatura o
mostrar un menú
```



1. Introducción

Ventajas

- Reduce la complejidad de un programa.
- Reutilización del código
- Mejora la claridad y legibilidad del código
- Mejora el diseño y la depuración de los programas
- Facilita el mantenimiento de los programas
- Permite la multiprogramación por módulos con programadores independientes para después integrarlos

2. Características

Independencia funcional:

- ✔ Un módulo debe realizar una única tarea.
- ✔ Comunicarse lo menos posible con el resto de módulos.
- ✔ Un módulo se debe dividir hasta que se consiga un nivel mínimo aceptable (independencia funcional).

La independencia funcional se puede medir según dos criterios:

Acoplamiento: Cómo dependen unos módulos de otros módulos.

Cohesión: Mide la relación entre las partes internas de un módulo encaminadas a realizar una **única función**.

2. Características

Acoplamiento Normal: Se refiere a cómo se pasan los datos entre módulos.

- **Datos**: se intercambian datos elementales
- **Estampado**: se intercambian estructuras compuestas.
- **Control**: se pasan datos para controlar su lógica (flags)

Acoplamiento Global: Comparten una zona de memoria (por ejemplo, una variable global)

Acoplamiento por Contenido: Un módulo lee o modifica los datos internos de otro.

Aconsejable: el Acoplamiento Normal Datos)

2. Características

Cohesión:

- **Funcional**: Importa el orden y realizan una única función.
- **Secuencial**: La salida de una tarea es la entrada de otra. Realizan más de un función.
- **Comunicacional**: Comparten datos de E/S. No importa orden.
- **Procedural**: Actividades distintas y no relacionadas.
- **Temporal**: Comparten instante de tiempo en que se realizan, no importa orden y no comparten datos.
- **Lógica**: Actividades de la misma categoría llamadas desde fuera del módulo
- **Casual**: Tareas diferentes, no relacionadas.

Aconsejable Cohesión: Funcional, Secuencial o Comunicacional.

2. Características

Un módulo o subprograma se compone de dos partes bien diferenciadas: La interfaz y la implementación.

- La interfaz es la parte que obtiene los datos de Entrada y facilita los datos de salida.
- La implementación es la parte donde el módulo realiza operaciones.

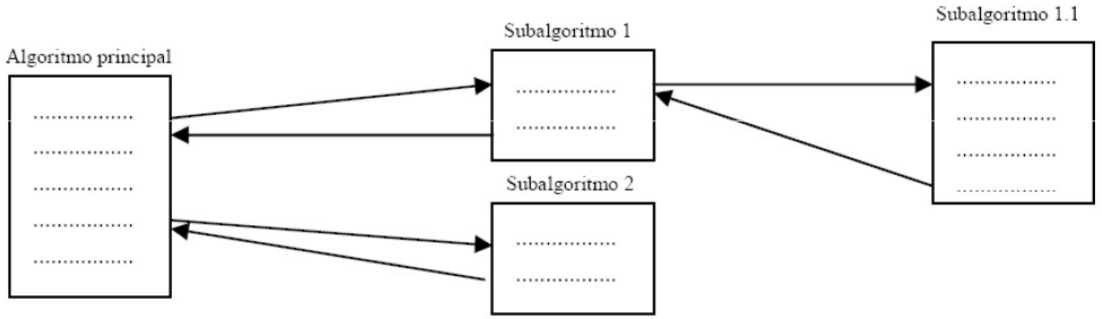
Desde el exterior, un módulo es una **caja negra.** A partir de unos datos muestra unos resultados pero no se sabe cómo lo hace. Es el concepto de Abstracción

3. Funciones

Siempre existe un **módulo principal (main)** que gestiona los módulos más sencillos (subalgoritmos).

Los subalgoritmos se usan desde el algoritmo principal. Éste decide cuándo debe ejecutarse cada subalgoritmo y con qué datos.

El módulo principal realiza llamadas y el resto devuelven los resultados al principal.



3. Funciones

Los subprogramas se llaman **funciones** y **procedimientos**.

Un subprograma realiza las mismas funciones que un programa:

1. Aceptan datos de entrada
2. Ejecutan instrucciones para procesar los datos
3. Devuelven unos resultados

El algoritmo principal va recogiendo los resultados de las funciones para generar la solución global al problema global.

3. Funciones

Diferencia entre **funciones** y **procedimientos:**

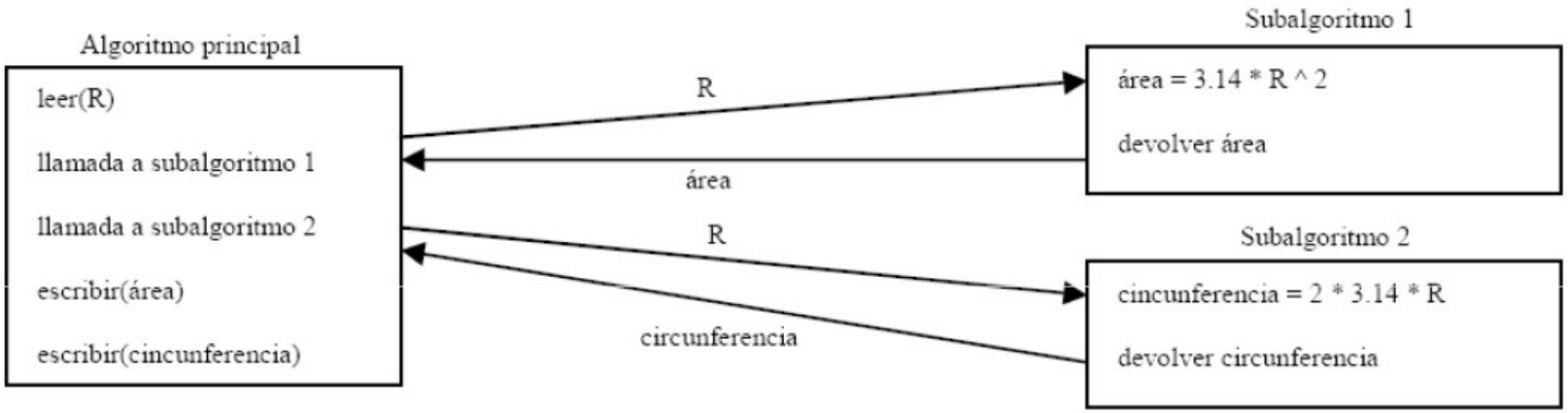
- Las **funciones** **devuelven un valor.**
- Los **procedimientos** ejecutan acciones o modifican estructuras de datos pero **no devuelven ningún valor**.

En Java, los procedimientos son idénticos a las funciones y se indican con **void** como dato devuelto.

Reciben los datos de entrada a través de los parámetros de la cabecera.

Ejemplo:

Algoritmo que pide el radio de una circunferencia al usuario y devuelve su área y perímetro.



Ejemplo:

```
final double PI = 3.141592;
Scanner sc = new Scanner(System.in);

public static double Area (double radio) {
    return PI * radio * radio;
}
public static double Circunferencia (double radio) {
    return 2 * PI * radio;
}
public static void main(String[] args) { // Programa principal
   double radio = sc.nextDouble();
   double area = Area(radio);//Llamada a la función Area
   double circ = Circunferencia(radio);//función Circunferencia

   System.out.println("El área es: "+area);
   System.out.println("La circunferencia es: "+circ);
}
```

3. Funciones

Estructura de una función Java

```
[acceso] [modificador] tipoDevuelto nombreMetodo([parámetros])
[throws listaExcepciones] {
   /* Bloque de instrucciones */
   [return valor;]
}
```

acceso: tipo de acceso al método (public, protected o private).

modificador : puede ser final y/o static. Las funciones independientes de un objeto son ***static***.

tipoDevuelto: tipo del valor que devuelve el método con la instrucción **return** o **void** si se trata de un procedimiento

nombreMetodo: nombre de la función o procedimiento.

Las **excepciones** las trataremos en su tema correspondiente

Ejemplo:

```
static int sumar(int a,int b) { // a y b son parámetros formales
   int c;
   c = a + b;
   return c;
}

public static void main(String[] args) { // Programa principal
   int num1 = 3, num2 = 5, suma = 0;

   suma = sumar(num1,num2);  //Llamada a la función sumar

   System.out.println("La suma de "+num1+" y "+num2+" es "+suma);
}
```

4. Parámetros

Los **parámetros** se pueden pasar de dos formas:

- Por **valor**: Los parámetros se usan de forma local realizando una copia del valor. Se destruyen fuera del subprograma.
- Por **referencia**: Se pasa la dirección de la Memoria en la que se almacena el valor. Puede ser modificado por el subprograma con efectos fuera del mismo

En Java todos los parámetros se pasan por valor, no por referencia.

Algunos objetos (por ejemplo vectores) almacenan referencias al objeto pero no son direcciones de memoria. Da la falsa sensación de paso por referencia

4. Parámetros

```
public class Intercambio {

  public static void interValor(double a, double b) {
    double aux;
    aux = a;
    a = b;
    b = aux;
    System.out.printf("interValor: a vale %f y b vale %f\n", a, b);
  }

  public static void interReferencia(double v[]) {
    double aux;
    for (int i = 0; i < (int) (v.length / 2); i++) {
      aux = v[i];
      v[i] = v[v.length - 1 - i];
      v[v.length - 1 - i] = aux;
    }

    System.out.printf("interReferencia: v vale %s\n", Arrays.toString(v));
  }
```

4. Parámetros

```
public static void main(String[] args) {
  double x = 5.0, y = 7.0;
  double[] vector = { 5.0, 7.0 };

  System.out.printf("Antes del intercambio: x es %f y el de y es %f\n", x, y);
  interValor(x, y);
  System.out.printf("Después del intercambio: x es %f y el de y es %f\n", x, y);

  System.out.printf("Antes de interReferencia vector vale %s\n",
                                                    Arrays.toString(vector));
  interReferencia(vector);
  System.out.printf("Después de interReferencia vector vale %s\n",
                                                    Arrays.toString(vector));

    }
}
```

```
Antes del intercambio: x es 5,000000 y el de y es 7,000000
interValor: a vale 7,000000 y b vale 5,000000
Después del intercambio: x es 5,000000 y el de y es 7,000000
Antes de interReferencia vector vale [5.0, 7.0]
interReferencia: v vale [7.0, 5.0]
Después de interReferencia vector vale [7.0, 5.0]
```

5. Sobrecarga de funciones

En java, una clase puede contener dos o más métodos con el mismo nombre si:

- Tienen diferente número de argumentos.
- Si el tipo de los argumentos es distinto, aunque tenga el mismo número de ellos

Para entender esta sencilla propiedad de java, lo mejor es contemplar el siguiente ejemplo de dos métodos sobrecargados.

5. Sobrecarga de funciones

Veamos un método sobrecargado por número de argumentos:

```
public void unMetodo() {
//Código del método
}
public void unMetodo(int numero) {
//Código del método
}

//Dos métodos con distinto número de argumentos:
unMetodo() // Ejecuta el primer procedimiento
unMetodo(5) // Ejecuta el segundo procedimiento
```

5. Sobrecarga de funciones

Veamos un método sobrecargado por tipo de argumentos:

```
public void otroMetodo(int entero) {
//Código del método
}
public void otroMetodo(float real) {
//Código del método
}

//Dos métodos con distinto número de argumentos:
otroMetodo(15) // Ejecuta el primer procedimiento
OtroMetodo(12.5) // Ejecuta el segundo procedimiento
```

6. Librerías

Una **librería**, también conocida como **biblioteca (library)**, es un conjunto de funciones y/o procedimientos (métodos):

- Pueden ser invocados y usados desde otro programa.
- No tienen la estructura de un programa independiente.
- Son una colección de utilidades o funcionalidades agrupadas por algún criterio.
- También conocidas como API (Application Program Interface) ya que ofrecen una interfaz para usar pero esconden su implementación.

En java se añaden con **import**

6. Librerías

Las librerías son especialmente útiles para:

- Reutilización de código.
- Ocultar la implementación.

Ejemplos librerías o bibliotecas:

- Operaciones de E/S: teclado, pantalla, ficheros, etc.
- Operaciones de cadenas: comparar, copiar, concatenar, obtener carácter, buscar, etc.

6. Librerías

```
package libreria; //Se encuentra en la carpeta libreria
import java.util.Scanner;

public class Vector {
    public static String[] rellenaVectorStr(int numElem) {
        Scanner sc = new Scanner(System.in);
        String[] vecStr = new String[numElem];
        for (int i = 0; i < numElem; i++) {
            System.out.printf("Introduce el elemento %d:",i);
            vecStr[i]=sc.nextLine();
        }
        sc.close();
        return vecStr;
    }
}
```

6. Librerías

```
import libreria.Vector;
import java.util.Arrays;

public class PruebaLibreriaV {
    public static void main(String[] args) {
        String[] vectorS;
          //Llamamos a la función para rellenar el vector
        vectorS=Vector.rellenaVectorStr(5);
        System.out.println(Arrays.toString(vectorS));
    }
}
```

6. Librerías

Completa la librería **Vector** con las siguientes funcionalidades que deberás ir invocando desde PruebaLibreriaV:

Operaciones con vector de **String**:

- Rellenar vector de String con datos del usuario (la del ejemplo)
- Obtener el elemento en la posición indicada del vector indicado
- Comprobar si dos vectores son iguales
- Obtener el reverso del vector (el primer elemento aparece el último)
- Modificar elemento del vector en la posición Indicada
- Imprimir el vector en horizontal
- Imprimir el vector en vertical

6. Librerías

Completa la librería **Vector** con las siguientes funcionalidades que deberás ir invocando desde PruebaLibreriaV:

Operaciones con vector de **Int**:

- Rellenar vector de Int pidiendo número y elementos por teclado
- Obtener el elemento en la posición indicada del vector indicado
- Comprobar si dos vectores son iguales
- Obtener el reverso del vector (el primer elemento aparece el último)
- Modificar elemento de la posición indicada del vector
- Sumar todos los elementos de un vector por N

6. Librerías

Completa la librería **Vector** con las siguientes funcionalidades que deberás ir invocando desde PruebaLibreriaV:

Operaciones con vector de **Double**:

- Rellenar vector de Int pidiendo número y elementos por teclado
- Modificar elemento de la posición indicada del vector
- Sumar todos los elementos de un vector por N
- Sumar los elementos de dos vectores