Unidad 3: Descomposición Funcional

Funciones, procedimientos, paso de parámetros y modularidad

Helder Fernandez Guzman

Universidad Mayor de San Simón– Introducción a la Programación

2/2025

Objetivos de la sesión

- Comprender la descomposición funcional de un problema en subproblemas más pequeños.
- Diferenciar entre **funciones** y **procedimientos**.
- Aplicar el paso de parámetros: **por valor** y **por referencia**.
- Reconocer **precondiciones** y **postcondiciones**.
- Identificar los principios de **modularidad**: cohesión y acoplamiento.

Motivación

- Resolver un problema complejo es más fácil si lo dividimos en partes más pequeñas.
- Ejemplo cotidiano:
  - Armar un **rompecabezas** pieza por pieza.
  - Cocinar un menú preparando cada plato por separado.
  - Construir una casa por etapas (cimientos, paredes, techo).
- En programación, estas piezas son las **funciones** y **procedimientos**.

Qué es la Descomposición Funcional

Definición:
Dividir un problema complejo en **subproblemas manejables** utilizando **funciones** y **procedimientos**.

- Facilita la comprensión y prueba de los programas.
- Relacionado con el principio de **modularidad**.
- Permite reutilizar código en diferentes contextos.

Funciones vs. Procedimientos

PSeInt

```
Funcion suma(a, b)
    Retornar a + b
FinFuncion

Proceso mostrarMensaje()
    Escribir "Hola!"
FinProceso
```

Java

```
int suma(int a, int b) {
    return a + b;
}

void mostrarMensaje() {
    System.out.println("Hola!");
}
```

Paso de Parámetros

- **Por valor:** se envía una copia del dato (en Java para tipos primitivos).
- **Por referencia (concepto):** el llamado puede modificar el argumento original.

PSeInt (por referencia)

```
SubProceso incrementar( Por Referencia x )
    x <- x + 1
FinSubProceso
Proceso main()
    n <- 5
    incrementar(n)
    Escribir n  // Imprime 6
FinProceso
```

Java (por valor en primitivos)

```
static void incrementar(int x) {
    x = x + 1; // No afecta al argumento original
}
public static void main(String[] args) {
    int n = 5;
    incrementar(n);
    System.out.println(n); // Imprime 5
}
// Nota: El "por referencia" en Java
// se verá más adelante con arreglos u objetos.
```

Condiciones de Entrada y Salida

- **Precondiciones:** lo que debe cumplirse antes de ejecutar la función.
- **Postcondiciones:** lo que se garantiza después de la ejecución.

Python

```python
import math

def raiz_cuadrada(x):
    # Precondición: x >= 0
    assert x >= 0
    y = math.sqrt(x)
    # Postcondición: y * y == x (aprox.)
    return y
```

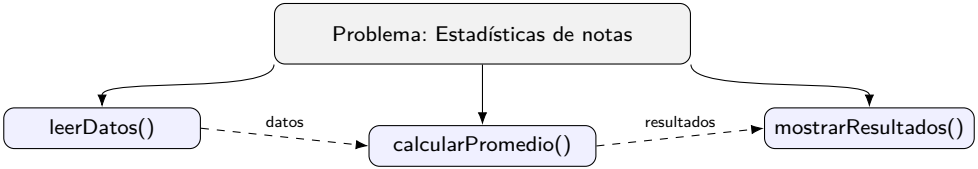
Java

```java
/** Calcula la raíz cuadrada de un número.
 * Precondición: x >= 0 */
double raizCuadrada(double x) {
    if (x < 0) {
        throw new IllegalArgumentException(
            "x debe ser >= 0");
    }
    return Math.sqrt(x);
}
```

Principios de Modularidad

- **Cohesión:** cada módulo debe encargarse de una sola tarea específica.
- **Acoplamiento:** reducir las dependencias entre módulos.
- **Beneficios:**
  - Programas más fáciles de leer y entender.
  - Reutilización de funciones y procedimientos.
  - Simplifica la depuración y las pruebas.

Modularidad: esquema visual



Cohesión alta (una tarea) Acoplamiento bajo (solo datos necesarios)

Cohesión alta y acoplamiento bajo

- **Cohesión alta:** cada función hace *una sola cosa*.
- **Acoplamiento bajo:** las funciones dependen lo menos posible entre sí.

Ejemplo con problemas

```
1 void procesarNotas() {
2     int n1 = 70, n2 = 85, n3 = 90;
3     // Calcula y muestra en el mismo módulo
4     double prom = (n1 + n2 + n3) / 3.0;
5     System.out.println("Prom: " + prom);
6 }
```

Ejemplo modular

```
1  static double promedio(int a, int b, int c) {
2      return (a + b + c) / 3.0; // Cohesión alta
3  }
4
5  static void mostrarPromedio(int a, int b, int c)
       {
6      System.out.println("Prom: " + promedio(a,b,c)
           );
7      // Acoplamiento bajo: recibe datos como
           parámetros
8  }
9
10 public static void main(String[] args) {
11     int n1 = 70, n2 = 85, n3 = 90;
12     mostrarPromedio(n1, n2, n3);
13 }
```

Diseño modular: **promedio()** tiene cohesión alta, **mostrarPromedio()** depende sólo de parámetros (bajo acoplamiento).

Buenas prácticas en descomposición

- Mantén las funciones **cortas y específicas**: cada una debe resolver un solo subproblema.
- Usa **nombres descriptivos** para funciones y parámetros.
- Define claramente las **entradas (precondiciones)** y **salidas (postcondiciones)**.
- Evita repetir código: extrae funciones comunes para reutilizarlas.
- Documenta con comentarios o JavaDoc qué hace cada módulo.
- Revisa la **cohesión** (alta) y el **acoplamiento** (bajo).

Ejemplo: Promedio y máximo de 3 notas

PSeInt

```
Funcion promedio(a, b, c)
    Retornar (a + b + c) / 3
FinFuncion

Funcion maximo(a, b, c)
    m <- a
    Si b > m Entonces m <- b FinSi
    Si c > m Entonces m <- c FinSi
    Retornar m
FinFuncion

Proceso main()
    Leer n1, n2, n3
    Escribir "Prom:", promedio(n1,n2,n3)
    Escribir "Max:", maximo(n1,n2,n3)
FinProceso
```

Java

```
static double promedio(int a, int b, int c) {
    return (a + b + c) / 3.0;
}

static int maximo(int a, int b, int c) {
    int m = a;
    if (b > m) m = b;
    if (c > m) m = c;
    return m;
}

public static void main(String[] args) {
    int n1 = 70, n2 = 85, n3 = 90;
    System.out.println("Prom: " + promedio(n1,n2,
        n3));
    System.out.println("Max: " + maximo(n1,n2,n3)
        );
}
```

Ejercicios propuestos (1520 min)

Resuelva cada problema usando funciones (sin arreglos ni objetos).

1. **Conversión de temperaturas:** leer C y mostrar F y K. Use `celsiusAFahrenheit(c)` y `celsiusAKelvin(c)`.
2. **Factorial:** implemente `factorial(n)` como función. Luego, un procedimiento `mostrarFactorial()` que lea y muestre el resultado.
3. **Promedio y máximo de 3 números:** implemente `promedio(a,b,c)` y `maximo(a,b,c)`.
4. **Áreas de figuras:** funciones como `areaTriangulo(b,h)` y `areaCirculo(r)`.

(Primero en PSeInt, luego en Java/Python).

Cierre de la unidad

- La **descomposición funcional** divide un problema en subproblemas manejables.
- Las **funciones** devuelven resultados, los **procedimientos** ejecutan acciones.
- El **paso de parámetros** puede ser por valor o por referencia.
- Las **precondiciones y postcondiciones** garantizan un uso correcto de las funciones.
- La **modularidad** mejora la legibilidad, la prueba y la reutilización del código.

Próximo paso: estructuras de control avanzadas combinadas con modularidad.