TEMA 3

SUBPROGRAMACIÓN

Subprogramas

3.1. Estructura en subprogramas

3.2. Subprogramas con parámetros

3.3. Vigencia y ámbito

3.4. Aspectos metodológicos

Objetivos

- Exponer las principales ventajas de la descomposición de programas en subprogramas
- Presentar la correcta construcción de programas y de subprogramas

3.1 Estructura en subprogramas

- Para solucionar un problema complejo se usa la estrategia de **divide y vencerás**, que consiste en partir el problema en subproblemas más simples. Este paso se repite con cada subproblema hasta que sus soluciones sean evidentes (**diseño descendente**).
- Para solucionar cada subproblema se realizan **subalgoritmos**. La mayoría de los lenguajes de programación permiten programar estos subalgoritmos por medio de **subprogramas.**

3.1 Estructura en subprogramas

- El problema principal se soluciona con el algoritmo principal (el primer paso de la división).
- El programa principal correspondiente se encarga de ensamblar (**llamar o invocar**) correctamente los distintos subprogramas y de intercambiar información entre ellos.
- Esta característica permite que los algoritmos sean simples, modulares y reutilizables.

3.1 Estructura en subprogramas

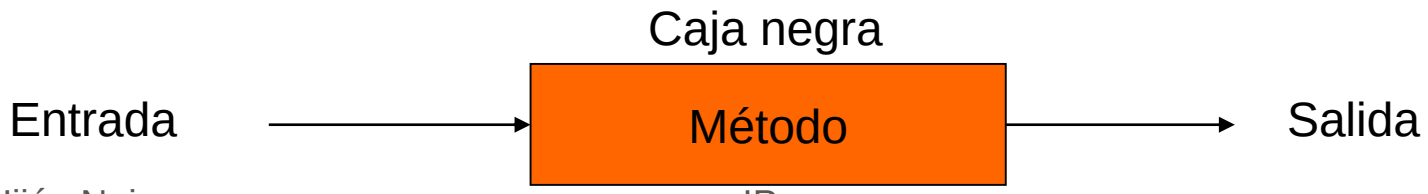
- **Problemas al escribir programas**:
  - Código fuente repetido
  - Falta de estructuración del código fuente
  - Difícil reutilización en otros programas
- **Solución**: **subprogramas**
  - pueden ser utilizados desde distintos puntos de un programa (evitan la repetición de código)
  - un subprograma soluciona una parte del problema inicial (facilita la estructuración)
  - Pueden ser usados en otros programas.

3.1 Estructura de subprogramas

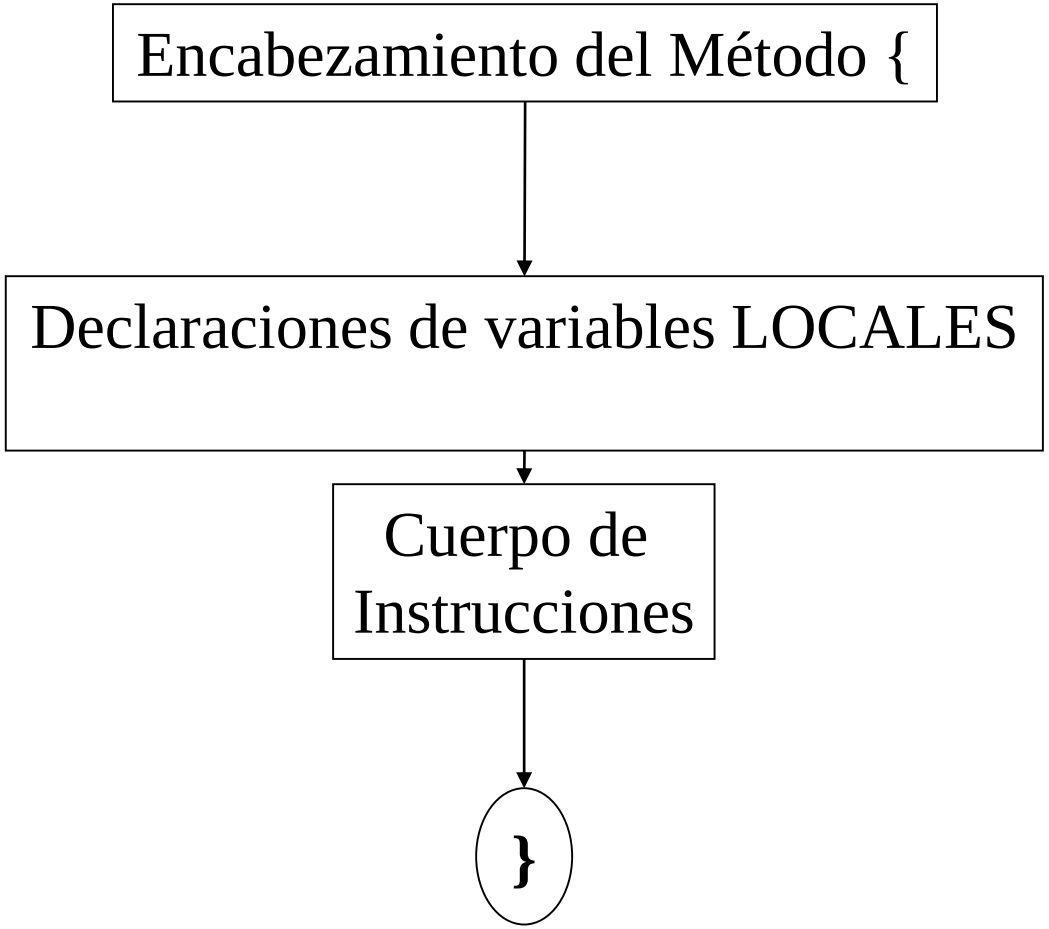
- En Java para crear un subprograma se utiliza un método.
  - Puede contener **expresiones e instrucciones** definidas por el programador.
    - Es una operación que puede tomar, cero, uno o más valores denominados **parámetros** y produce un valor denominado **resultado**. Es similar a la función matemática.
    - Ejecuta un proceso especifico.

SUBPROGRAMAS

- Existen dos puntos de vista:
  - El **programador del método**. Se encargará de diseñarlo y declararlo
    - Datos de entrada exteriores al subprograma, datos que va a devolver (exportar), tarea a realizar y como va realizarla.
  - El que **va a llamar a método**. Se encargará de llamarlo (INVOCACIÓN). Sólo tiene que tener claro:
    - Datos tiene que darle y qué le va a devolver (modelo de la caja negra).



Estructura sintáctica de subprogramas



Estructura sintáctica:

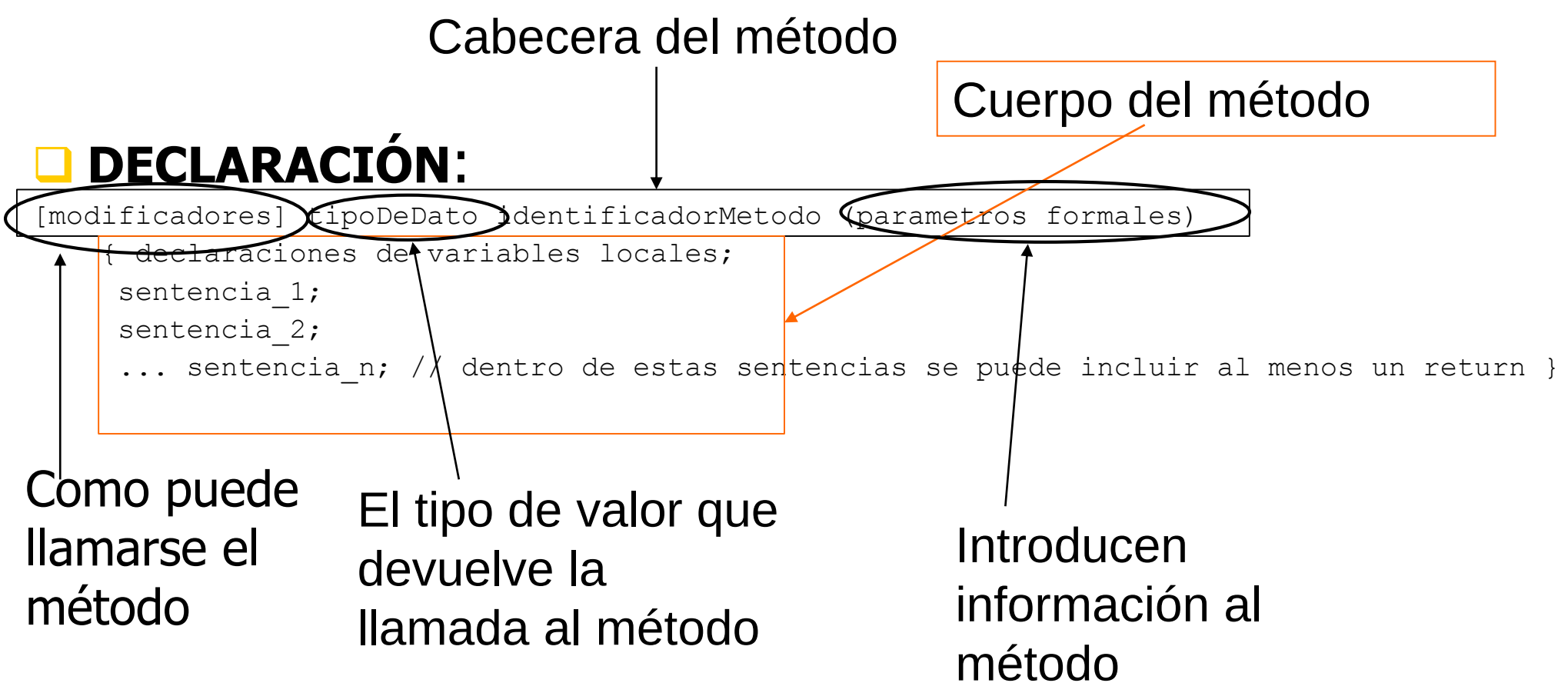
MÉTODOS

- Un método es un trozo de código que puede ser llamado o invocado por el programa principal o por otro método para realizar alguna tarea específica. generalmente tomando unos valores de entrada (argumentos) y devolviendo un único valor.
- Es llamado por su nombre o identificador seguido por una secuencia de parámetros o argumentos entre paréntesis.
- Puede devolver un valor simple al programa que lo llama
- El tipo de dato devuelto por la sentencia *return* debe coincidir con el tipo de dato declarado en la cabecera del método

DECLARACIÓN:

```
[modificadores] tipoDeDato identificadorMetodo (parametros formales)
        declaraciones de variables locales;
       sentencia_1;
       sentencia_2;
       ... sentencia_n; // dentro de estas sentencias se puede incluir al menos un return }
```

Estructura sintáctica:



3.2 Métodos con parámetros

- TIPOS DE PARÁMETROS
  - Dependiendo del lugar donde se encuentren se clasifican en:
  - **Parámetros formales** o ficticios: Se encuentran en la definición (cabecera) del subprograma.
    - `public static double tanDeGrados(double a){`
  - **Parámetros actuales** o reales: Se encuentran en la llamada al método.
    - `double tangente =  tanDeGrados(60);`

Métodos con parámetros

- TIPOS DE PARÁMETROS:
  - Dependiendo del flujo de información o de la forma de **intercambiar información** entre el método y el resto de métodos o el programa principal *main* se clasifican en:

  - Parámetros de entrada (al método)

  - Parámetros de salida (del método)

Métodos con parámetros

- **PARÁMETROS DE ENTRADA:**

  También se conocen como **PARÁMETROS POR VALOR**:

  - Utilización: Parámetros de entrada.
  - Sintaxis: Igual que una declaración de variable
  - Ejemplo:

```
public static double tanDeGrados(double a){
```

Mecanismo de paso de parámetros por valor

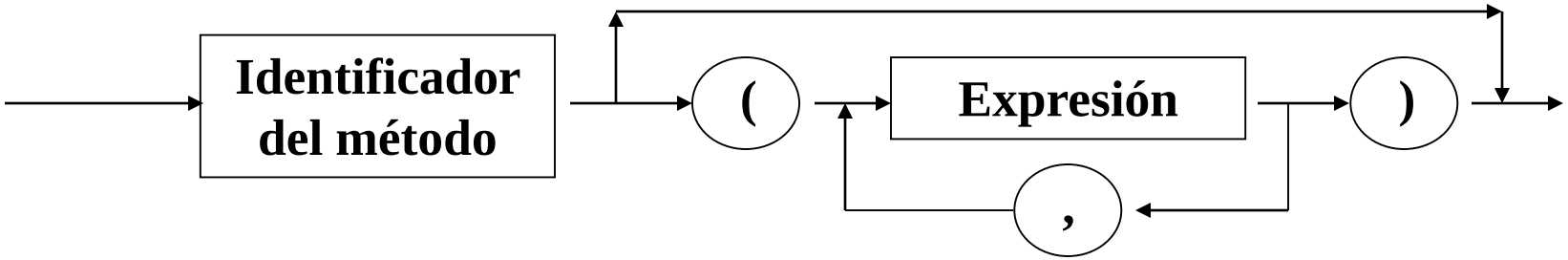
- **Semántica**:
  - se calcula el valor de los parámetros reales en la llamada (evaluando las expresiones correspondientes)
  - una copia de dicho valor se asigna a los parámetros formales del subprograma
  - el subprograma opera sobre esta copia
  - al finalizar el subprograma se pierde su estado de cómputo local, y cualquier cambio hecho en el parámetro formal **NO** quedará reflejado en el parámetro real

Mecanismo de paso de parámetros por valor

- **Restricciones**:
  - Los parámetros reales pueden ser expresiones, variables o constantes.
  - Los parámetros formales y reales han de ser de tipo **asignación-compatibles**

Estructura sintáctica: Llamadas

- **LLAMADAS A MÉTODOS**:
  - igual que los **métodos predefinidos**

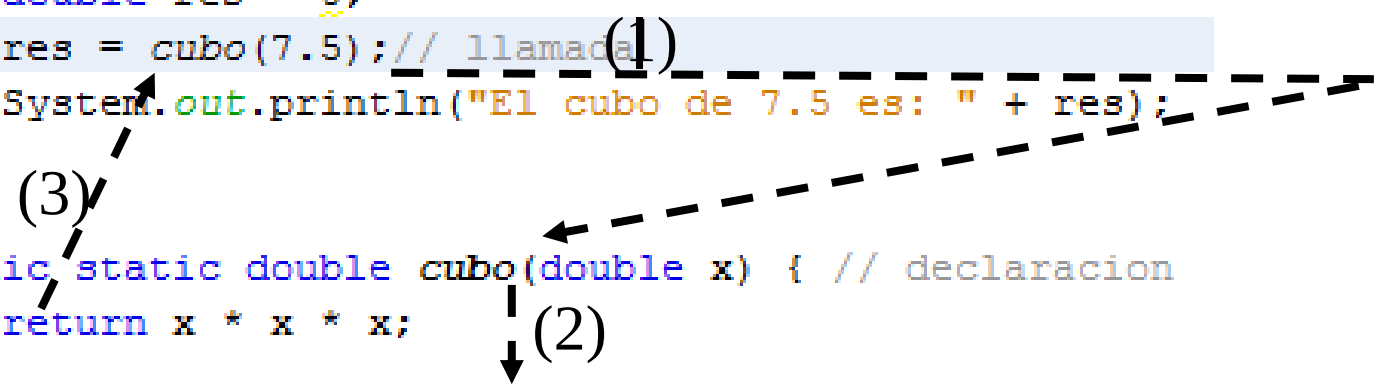


Semántica: llamada a un método

```
public class Metodos1 {

    public static void main(String[] args) {
        double res = 0;
        res = cubo(7.5);// llamada
        System.out.println("El cubo de 7.5 es: " + res);
    }

    public static double cubo(double x) { // declaracion
        return x * x * x;
    }
}
```



`res` ha de ser de un tipo compatible con `el tipo del método`

Resultado de un método: return

- **Sentencia RETURN**
  - la sentencia **`return`** se emplea para salir de la secuencia de ejecución de las sentencias de un método.
  - Opcionalmente pueden devolver un valor
  - Tras la salida se vuelve al lugar de llamada de dicho método.
  - Convenio: solo pondremos uno y al final del método cuando devuelvan algo
  - El tipo de retorno de un método se especifica en la declaración del método
  - Cualquier método declarado como *void* no devolverá ningún valor.
  - El tipo de dato del valor de retorno deberá coincidir con el tipo de retorno de la declaración del método.
- **SINTAXIS**:

```
return valorRetorno;
```

Subprogramas con parámetros

MÉTODOS CON PARÁMETROS

return expresión;

```
public class Metodos1 {

    public static void main(String[] args) {
        double res = 0;
               res = producto(7,3);// llamada
        System.out.println("El producto de 7 X 3 es: " + res);
    }

    public static int producto (int a, int b) { // declaracion
        int c;
        c = a*b;
        return c;
    }
}
```

Métodos – Tipo void, no devuelve nada

- **Ejemplo 1a:**

```
public class Metodos {

    static void imprimirPrimos(int max, int num) {
        num = 2;
        System.out.println("1\n2");
        for (int i = 3; i <= max; i++) {
            int divisor = 2;
            while (i % divisor != 0 && divisor < i - 1) {
                divisor++;
            }
            if (divisor == i - 1) {
                System.out.println(i);
                aux = num++;
            }
        }
    }

    public static void main(String[] args) {
        int maximo = 25, numero = 0;

        System.out.println("Números primos hasta " + maximo + ": ");
        //No obtendremos el valor que queremos
        imprimirPrimos(maximo, numero);
        System.out.println("Total " + numero + " primos"); // Mal"
    }
}
```

- **Salida:**

```
Números primos hasta 25:
1
2
3
5
7
11
13
17
19
23
Total 0 primos
```

Métodos – Se devuelve un entero

Ejemplo 1 b

```
public class Metodos {

    public static int imprimirPrimos(int max, int num) {
        int aux = 0;
        num = 3;
        //System.out.println("1\n2"); Es lo mismo que las 2 sentencias siguientes
        System.out.println("1");
        System.out.println("2");
        for (int i = 3; i <= max; i++) {
            int divisor = 2;
            while (i % divisor != 0 && divisor < i - 1) {
                divisor++;
            }
            if (divisor == i - 1) {
                System.out.println(i);
                aux = num++;
            }

        }
        return aux;
    }

    public static void main(String[] args) {
        int maximo = 25, numero = 0, n = 0;
        System.out.println("Números primos hasta " + maximo + ": ");

        System.out.println("Total " + imprimirPrimos(maximo, numero) + " primos");
    }
}
```

Salida:

```
run:
Números primos hasta 25:
1
2
3
5
7
11
13
17
19
23
Total 10 primos
BUILD SUCCESSFUL (total time: 0 seconds)
```

Métodos – se devuelve un real

- **Ejemplo 2:**

```
public class Metodo1 {

public static void main (String [] args){
 System.out.println("El cubo de 7.5 es: " + cubo(7.5)); // llamada
 }
 public static double cubo (double x) { // declaracion
 return x*x*x;
 }
 }
```

- **Salida:**

```
run:
El cubo de 7.5 es: 421.875
BUILD SUCCESSFUL (total time: 0 seconds)
```

Métodos – no devuelve nada

- **Ejemplo 3:** Dibujar un cuadrado pidiendo el lado del cuadrado

cuadrado(4);

```
public static void cuadrado(int lado) {
    int numeroasteriscos = lado;

    //Dibujamos la parte de arriba del cuadrado
    for (int cont = 0; numeroasteriscos > cont; cont++) {
        System.out.print("*");
    }
    System.out.println("");

    //Usamos un bucle anidado para dibujar los asteriscos del medio
    for (int cont = 1; (numeroasteriscos - 2) >= cont; cont++) {
        System.out.print("*");
        //Este bucle dibuja los espacio entre el primer y ultimo asterisco
        for (int i = 0; (numeroasteriscos - 2) > i; i++) {
            System.out.print(" ");
        }
        System.out.print("*");
        System.out.println("");
    }

    //Dibujamos la parte de abajo del cuadrado
    for (int cont = 0; numeroasteriscos > cont; cont++) {
        System.out.print("*");
    }
}
```

Salida:

```
****
*  *
*  *
****BUILD SUCCESSFUL (total time: 0 seconds)
```

Estructura sintáctica: Declaraciones

- **DECLARACIONES LOCALES**
  - Idénticas que las del main:

    constantes
    variables
    métodos

    } Propios del método. Sólo desde él se accede a ellos

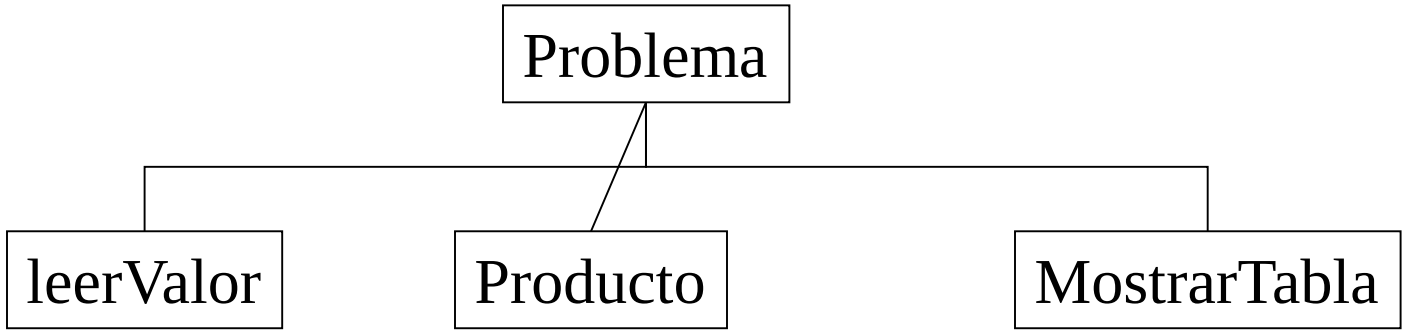
  - Estas definiciones y los parámetros constituyen los **elementos locales** del subprograma

Estructura sintáctica: Cuerpo

- **CUERPO DE INSTRUCCIONES**
  - secuencia de instrucciones
  - implementa el algoritmo que resuelve el problema para el que se definió el método
  - las instrucciones del cuerpo se ejecutan únicamente con cada llamada al método

Ejemplo: mecanismos de paso de parámetros

- Problema:
  - *Leer* un valor, *Mostrar la tabla* de multiplicar para ese valor, llamando a un método que calcule el *producto* de dos números



Ejemplo: mecanismos de paso de parámetros I

```java
import java.io.*;
import java.util.Scanner;

public class PruebaTabla {

    public static void main(String[] args) throws IOException {
        int val = 0;
        val = leer();
        tabla(val); // ejemplo de llamada
    }

    public static void tabla(int n) { // de tipo void
        System.out.println("Tabla de multiplicar del numero " + n);
        for (int i = 0; i <= 10; i++) {
            System.out.println(n + " x " + i + " = " + producto(n, i));
        }
    }

    public static int leer() throws IOException {
        Scanner entrada = new Scanner(System.in);
        int valor;
        System.out.println("Teclea un numero del 1 al 10 seguido de ENTER");
        valor = entrada.nextInt();;
        return valor;
    }

    public static int producto(int a, int b) {
        return a * b;
    }
}
```

Salida

```
Teclea un numero del 1 al 10 seguido de ENTER
9
Tabla de multiplicar del numero 9
9 x 0 = 0
9 x 1 = 9
9 x 2 = 18
9 x 3 = 27
9 x 4 = 36
9 x 5 = 45
9 x 6 = 54
9 x 7 = 63
9 x 8 = 72
9 x 9 = 81
9 x 10 = 90
BUILD SUCCESSFUL (total time: 4 seconds)
```

3.3 Vigencia y ámbito

- **ASPECTOS**
  - Estructura de bloques
  - Vigencia y ámbito identificadores

Estructura de bloques:

- **bloques**:
  - Un bloque es una estructura de programa que agrupa sentencias. Los bloques comienzan con una llave de apertura ({) y terminan con una llave de cierre (}). Un bloque puede estar dentro de otro bloque.

Estructura de bloques: Ejemplo I

```
public class Bloque {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        {
            int suma, x;
            x = 1;
            suma = 0;
            while (x <= 10) {
                suma += x;
                x++;
            }
            System.out.println("La Suma es: " + suma);
        }
    }
```

Estructura de bloques: Ejemplo II

```
public class Ambito {

    public static void main(String[] args) {
        int i;
        for (i = 5; i >= -5; i--) {
            System.out.println(i + " es positivo: " + esPositivo(i));
            System.out.println("    y espar: " + esPar(i));
        }
    }

    public static boolean esPar(int p) {
        if (p % 2 == 0) {
            return true;
        } else {
            return false;
        }
    }

    public static boolean esPositivo(int x) {
        if (x < 0) {
            return false;
        } else {
            return true;
        }
    }
}
```

ALCANCE DE LAS VARIABLES

- VARIABLES **LOCALES A UN BLOQUE**
  - se declaran en el bloque

    Ejemplo: `p` es var. local de `EsPar`

    `x` es var. local de `EsPositivo`
- VAR. **NO-LOCALES A UN BLOQUE**
  - se declaran en un bloque **exterior al bloque interior**

    Ejemplo: `n` es var. no-local al `bloque for`

```
public static void tabla(int n) { // de tipo void
    System.out.println("Tabla de multiplicar del numero " + n);
    for (int i = 0; i <= 10; i++) {
        System.out.println(n + " x " + i + " = " + producto(n, i));
    }
}
```

Vigencia y Ámbito

- **VIGENCIA** (o vida) de un objeto:
  - los bloques del programa en los que el objeto "**existe**" (i.e. tiene espacio de memoria asignado)
  - un objeto es **vigente** en el **bloque** en el que está definido **y** en todos los **bloques interiores** a él
- **ÁMBITO** (o visibilidad) de un identificador:
  - los bloques en los que se puede **acceder** a un objeto
  - el ámbito de un identificador es el **bloque** del método en el que está definido, incluyendo todos los **bloques interiores** a él

3.4 Aspectos metodológicos

- **Desarrollo o diseño descendente**
  - Proceso de **descomposición** y **refinamiento progresivo** de un problema en subproblemas más pequeños, hasta llegar a un problema fácilmente resoluble por la mente humana.

- **Refinamiento sucesivo**
  - **Descomposición por niveles** cada vez más específicos (soluciones más detalladas).
    - Clase Main
    - Métodos del siguiente nivel
    - Métodos de de niveles sucesivos

Desarrollo descendente

- **Desarrollo descendente y subprogramas**
  - Se suele **definir** un **método** para cada **subproblema**
  - La precondición y la postcondición del método especifican el (sub-)problema que resuelve
  - El cuerpo del método implementa el algoritmo que resuelve el subproblema
  - Los diagramas estructurales ilustran la descomposición del problema en subproblemas/ subprogramas

Recomendaciones Técnicas

- Utilizar funciones y procedimientos en el marco del **desarrollo descendente**.

- Aspectos relevantes:
  - **Encapsulación**

Encapsulación

- **Principio de máxima localidad**
  - Los objetos particulares y necesarios para un método(especialmente las variables) deben ser locales al mismo
- **Principio de autonomía de los métodos**
  - La comunicación con el exterior debe realizarse exclusivamente mediante parámetros, evitándose dentro de los métodos toda referencia a objetos globales

Subprograma
=
Caja Negra

Variables externas

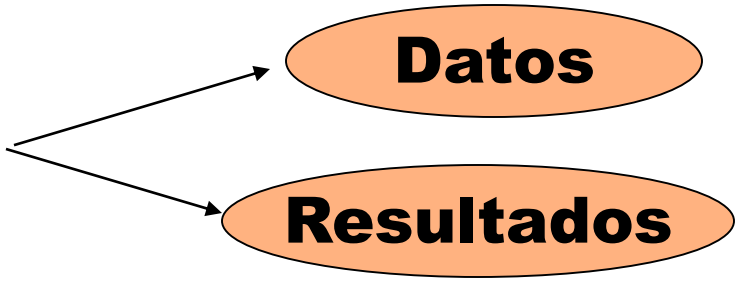
- Un **subprograma** debe **usar** sus **variables locales**
- Un **subprograma NO** debe usar **variables externas**
- Para **usar** el valor de variables externas en subprogramas: pasarlas como **parámetros por valor**

Variables externas

- Queda **absolutamente prohibido** modificar el valor de las variables externass directamente en un subprograma (efectos laterales)

Recomendaciones

- Para utilizar de manera correcta la subprogramación debe seguir los siguientes pasos:
- **1- Analizar el problema**



- **2- Diseñar un algoritmo**
  - **2-1 Diseñar el programa principal, utilizando subprogramas (instrucciones o funciones nuevas).**
  - **2-2 Diseñar los subprogramas (métodos).**

Recomendaciones

- Para utilizar de manera correcta la subprogramación debe seguir los siguientes pasos:
- **3- Implementar el programa.**
  - **3-1 Escribir el programa y las cabeceras de todos los métodos.**
  - **3-2 Escribir un método y probarlo.**
  - **3-3 Si quedan subprogramas ir al 3-2**
- **4- Probar el programa completo.**

Sugerencias MÉTODOS

- Si al tratar de realizar una tarea, el método se extiende, se considerará la división del método.
- Cualquier tarea que se efectúa más de una vez en un programa, debe ser un método.

Sugerencias MÉTODOS

- Los métodos deben ser lo más generales posibles, es decir, servir para el mayor número de entradas.

- Los métodos deben ser pequeños, con un tamaño máximo de 20 líneas aproximadamente, para facilitar su depuración.

3.5 Análisis de Corrección

- **Análisis de corrección**
  - Corrección sintáctica
  - Corrección semántica: depuración
  - Corrección semántica: verificación formal

Análisis de Corrección

- **Corrección sintáctica**:
  - del subprograma
    - diagramas sintácticos (i.e. similar al programa principal)
  - de la llamada
    - los parámetros formales y los reales tienen que coincidir en número y tipo

Depuración

- **Depuración de la llamada**
  - depurar la llamada como una sólo instrucción/ expresión (definida por el programador)
  - se ejecuta el cuerpo completo del subprograma, y se examina el estado de cómputo resultante
- **Depuración del método**
  - depurar las instrucciones del subprograma una por una
  - se examinan sucesivamente los estados de cómputo locales del subprograma
- **En el depurador de NetBeans:**
  - Menu "*Debug*",

Proceso de corrección y depuración

- **Proceso**:
  - Los métodos se deben poder depurar, verificar y probar independientemente
  - La construcción, depuración, verificación se hará de forma incremental
    - **Construcción**: desde los niveles de mayor abstracción a los de menor abstracción (diseño descendente)
    - **Depuración y verificación**: se suelen aplicar primero a los subprogramas de menor abstracción, para luego ir "ascendiendo"

Pruebas de subprogramas

- Se prueban a dos niveles:
  - Pruebas de "caja blanca"
  - Pruebas de "caja negra"

Pruebas de “caja negra”

- Pruebas en que se conoce sólo la interfaz
  - Caja negra (*black box*: caja opaca)
  - Se procura ejercitar cada elemento de la interfaz
- Algunas clases de pruebas
  - Cubrimiento → invocar todas las funciones (100%)
  - Clases de equivalencia de datos
  - Pruebas de valores límite

Pruebas de clases de equivalencia

- Particiones de equivalencia
  - Los datos se clasifican según las distinciones visibles en la interfaz del programa.
  - Ejemplo: EsPrimo: Entero → Booleano
    - Clase 1: primo ≥ 2 (2, 3, 5, 7, 11, ...)
    - Clase 2: no_primo ≥ 2 (4, 6, 8, 9, 10, ...)
    - Clase 3: valores singulares (0, 1)
    - Clase 4: no definido (-1, -2, ...)
- Casos de ensayo con datos de cada clase

Pruebas de valores límite

- Complemento a las particiones de equivalencia
- Varios casos de prueba por cada partición
  - Valores típicos, intermedios
  - Valores primero y segundo del rango
  - Valores penúltimo y último
  - Valores vecinos fuera del rango (en otra partición)
- Motivación
  - Los programadores se equivocan con más frecuencia al tratar los valores en la frontera (Ej: > en vez de ≥)

Pruebas de subprogramas

- Se prueba cada subprograma