PROGRAMACIÓN MODULAR

MÉTODOS EN JAVA

Programación Modular

- En Java toda la lógica de programación (Algoritmos) está agrupada en funciones o métodos.
- Un método es:

Un bloque de código que tiene un nombre, recibe unos parámetros o argumentos (opcionalmente), contiene sentencias o instrucciones para realizar algo (opcionalmente) y devuelve un valor de algún Tipo conocido (opcionalmente).

- La sintaxis global es:

Tipo_Valor_devuelto ***nombre_método*** **(** *lista_argumentos* **) {**
bloque_de_codigo;
}

- y la lista de argumentos se expresa declarando el tipo y nombre de los mismos (como en las declaraciones de variables). Si hay más de uno se separan por comas.

Por ejemplo:

```
int sumaEnteros ( int a, int b ) {
        int c = a + b;
        return c;
  }
```

- El método se llama sumaEnteros.
- Recibe dos parámetros también enteros: a y b.
- Devuelve un entero.
- En el ejemplo la claúsula **return** se usa para finalizar el método devolviendo el valor de la variable c.

El termino void

- El hecho de que un método devuelva o no un valor es opcional. En caso de que devuelva un valor se declara el tipo que devuelve. Pero si no necesita ningún valor, se declara como tipo del valor devuelto, la palabra reservada **void**. Por ejemplo:

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

- Cuando no se devuelve ningún valor, la claúsula **return** no es necesaria. En el ejemplo el método mostrar tampoco recibe ningún parámetro. No obstante los paréntesis, son obligatorios.

Uso de métodos

- Los métodos se invocan con su nombre, y pasando la lista de argumentos entre paréntesis. El conjunto se usa como si fuera una variable del Tipo devuelto por el método.

Por ejemplo:

```
int x;
   x = sumaEnteros(2,3);
```

- Aunque el método no reciba ningún argumento, los paréntesis en la llamada son obligatorios. Por ejemplo para llamar a la función mostrar, simplemente se pondría:

```
mostrar();
```

- Observad que como la función tampoco devuelve ningún valor no se asigna a ninguna variable. (No hay nada que asignar).

Métodos en Java

- Tipo devuelto por un método.
- Terminación de métodos: return.
- Parámetros de un método.
- Sobrecarga de métodos. Firma y prototipo de un método.

Definición de métodos en Java

Método: Sección de código autocontenida que pertenece a una clase y que define la parte del comportamiento del sistema correspondiente a dicha clase y a sus objetos.

Sintaxis:

```
[Modificadores] tipoDevuelto identificadorMetodo([argumentos])
{
// cuerpo del método
}
```

Ejemplo: **public int maximoDosNumeros(int n1, int n2)**

- *Los modificadores son las palabras reservadas **public**, **protected**, **private**,**final** y **static** (se verán en temas posteriores).*
- *Lo que aparece entre corchetes es opcional.*

Métodos en Java

Observaciones:

- *La tarea asignada a un método debe estar bien definida.*
- *El nombre de un método debe indicar exactamente lo que hace (mediante un verbo).*
- *Los métodos deben tener una longitud adecuada.*

  Métodos demasiado extensos hacen difícil su comprensión y depuración

Métodos en Java: ¿Dónde se declaran?

- *En Java no existen métodos fuera de las clases*
  → Todos los métodos se definen en el interior de una clase.
- *No es posible definir métodos dentro de métodos.*

```
class UnaClase {
   // variables.
   // métodos.
}
```

- *La clase es un tipo de datos definido por el usuario → Tipo Abstracto de Datos.*
- *La clase define:*
  → Los atributos de los objetos:
  → La interfaz que exponen los objetos.
  → El comportamiento de los objetos: su implementación.
- *Las clases se explicarán con detalle en temas posteriores.*

Tipo devuelto por un método

Tipo devuelto: Es el tipo del valor devuelto al objeto que invocó al método. Puede ser la palabra reservada **void**, *un tipo primitivo Java o una clase .*

Se devuelve un valor primitivo Java.	`double pow(double d1, double d2)`
No se devuelve nada.	`void pintarCirculo(float radio)`
Se devuelve una referencia a objetos.	`Coche cocheConMasMultas()` `String substring(int start, int end)`

Terminación de métodos: return.

Dos posibilidades:

Si el método devuelve `void` *(nada) el método termina cuando llega al final del método (cierra llaves) o cuando hace* `return`.

```
public void pintaCirculo(float r){
   // código para pintar el círculo
} // este es el final del método

public void saluda(String mensaje){
   if (!(primeraVez))
      return;  // aquí termina
   else
      System.out.println(mensaje);
}
```

Si el método devuelve algo distinto de `void` *entonces se devuelve el control mediante:*

`return expresión`

donde `expresión` *produce un valor cuyo tipo es el declarado en la cabecera del método.*

```
// Cálculo de x^y
int potencia(int x, int y){
   int contador = 0, resultado = 1;
   while (contador != y){
      resultado = resultado * x;
      contador++;
   }
   //aquí se devuelve el resultado.
   return resultado;
}
```


Terminación de métodos: return.

Cuando se hace `return` *se devuelve el control al invocador del método.*

Registro de Activación

Parámetros.
Valor de retorno.
Dirección de retorno.
Datos locales.

Es un error de compilación cuando se devuelve siendo `void` *el tipo de retorno.*

```
public void saluda(String mensaje){
   if (!(primeraVez))
      return false;  // error
   else
      System.out.println(mensaje);
}
```

Es un error de compilación no hacer `return` *cuando hay que devolver un valor de un tipo.*

```
int potencia(int x, int y){
   int contador =0, resultado = 1;
   while (contador != y){
      resultado = resultado * x;
      contador++;
   }
   // Error: No se retorna valor
}
```

Lista de parámetros de un método

La lista de parámetros (argumentos) es una lista de las declaraciones de parámetros que se pasan al método para su ejecución.

- La lista puede estar vacía pero los paréntesis son obligados:

public void suficiente() ***// Correcto***
public void pocaCosa ***// Error de sintaxis.***

- Es obligatorio indicar el tipo de cada parámetro:

double potencia (double x, double y) ***// Correcto***
double potencia (double x, y) ***// Error de sintaxis***

Parámetros formales y parámetros reales

- Parámetros formales: Los que se definen en la cabecera de la función.
- Parámetros reales: Los que aparecen en la llamada a la función. Deben corresponderse con los parámetros formales definidos en la cabecera.

Parámetros formales

```
int  f ( int  n, boolean t )
```

Los argumentos de entrada deben coincidir en número y tipo y aparecer en el mismo orden con los declarados en el método.

```
public static void main(String [] args){
   int k = 3;
   boolean flag = true;
   // Llamadas correctas:
   f(7, false);
   f(0, flag);
   f(k, flag);
   // Llamadas incorrectas:
   // No hay concordancia entre parámetros reales y formales
   f(true, d, flag);
   f(false, 7);
}
```

Parámetros reales

```
f(7, false);
```

Promoción de argumentos. Coerción.

Consideremos la siguiente cabecera de un método:

public static double sqrt(double d)

Y el siguiente uso del método:

```
int numero = 4;
System.out.println(Math.sqrt(numero));
```

Java promociona implícitamente el argumento de **int** a **double** antes de pasarlo a **sqrt()**, pasando **4.0** en lugar de **4**.

A esto se le llama *coerción de argumentos* y sigue determinadas *reglas de promoción*.

Las reglas de promoción especifican cómo los tipos pueden convertirse a otros sin que haya pérdida de información.

Reglas de promoción de argumentos.

Tipo	Promociones permitidas
double	ninguna (no hay tipo más grande que double)
float	double
long	float o double
int	long, float o double
char	int, long, float o double
short	int, long, float o double
byte	short, int, long, float o double
boolean	ninguna (ni true ni false son números)

- Estas reglas se aplican en la coerción de argumentos y en las expresiones que contienen dos o más tipos de datos.
- Los valores originales de las variables no se modifican: temporalmente se promocionan.
- Las conversiones a tipos más pequeños sólo es posible haciendo conversión explícita mediante *casting*. Puede perderse información.

Parámetros por valor y por referencia

Los parámetros de los métodos se pueden pasar por valor o por referencia

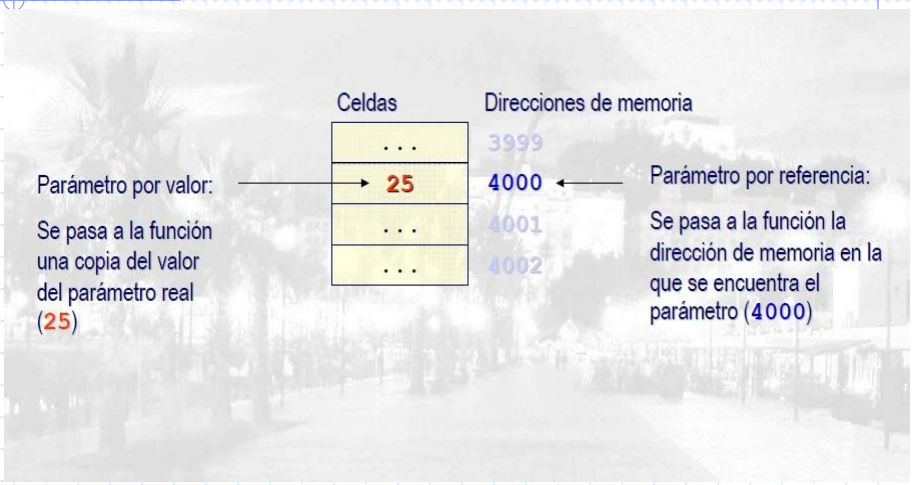
Paso de Parámetros por Valor:	*Paso de Parámetros por Referencia:*
Se le pasa al método una copia del valor.	*Se le pasa al método una referencia a los datos.*
Los cambios realizados dentro del método no afectan a la variable que se utilizó como parámetro.	*El método invocado puede modificar los datos del invocador.*
Ofrece seguridad.	*Se ahorra memoria (no se copian los datos).*

Algunos lenguajes ofrecen la posibilidad de elegir si los parámetros se pasan por valor o se pasan por referencia.

→ ***En `Java`, los tipos primitivos siempre se pasan por valor y los objetos siempre por referencia.***

→ ***Esto también ocurre para las variables del la sentencia `return`.***

Mecanismos de paso de parámetros



Paso de *arrays* como parámetros

- Los *arrays* son objetos y por tanto se pasan por referencia.
- Para pasar un *array* como parámetro escribimos el nombre del *array* en la lista de parámetros (sin corchetes)

```
int temperaturasHorarias[] = new int[24];
modificarArray(temperaturasHorarias);
```

- El nombre del *array* es en realidad una referencia a un objeto que contiene los elementos del *array.*

Paso de *arrays* como parámetros.

```
class Paso_Arrays_1 {

  static int a[] = {0, 1, 2, 3, 4};

  public static void modificarArray(int b[]){ // Por referencia
      for (int j = 0; j< b.length; j++ ) b[j] *= 2;
   }

  public static void modificarElemento(int e){ // Por valor
       e *= 2;
   }

  public static void main(String [] args){
     modificarArray(a); // después de llamada a = {0, 2, 4, 6, 8}
     modificarElemento(a[3]); // después de llamada a[3] = 6;
   }
}
```

Paso de *arrays* como parámetros

```
class Paso_Arrays_2 {

   int static a[] = {1, 2, 3};

  public static void modificarArray(int a[]){
     int [] b = {4, 5, 5};
    a[0] = b[0];
    b[1] = a[1];
    a = b;
     // ¿qué se imprime?
   for (int i = 0; i < a.length; i++) System.out.println(a[i]);
   }
  public static void main(String [] args){
   modificarArray(a);
   // ¿qué se imprime?
  for (int i = 0; i < a.length; i++) System.out.println(a[i]);
  }
}
```


Sobrecarga de métodos. Firma y prototipo de un método

Sobrecarga de métodos: Java permite definir métodos con el mismo nombre siempre que difieran en los argumentos de entrada.

- Un método se distingue del resto por su **firma.**

  Firma del método: nombre + argumentos de entrada.

- Puesto que los métodos sobrecargados tienen el mismo nombre, los métodos sobrecargados se diferencian por el número, tipo y orden de los argumentos en su lista de parámetros.

```
public class utilidadesMatematicas{
   int    square(int x) { return x*x; }
   double square(double x) { return x*x; }
}
```

El **prototipo de un método** es la firma más el tipo devuelto.

Los métodos sobrecargados no se distinguen por el prototipo sino por la firma.

Variables locales en los métodos

Declaración de variables locales

***[modificador]* tipoValorDevuelto nombre([*argumentos*])**
// declaración de variables locales
// enunciados
}

Las variables pueden ser de tipos primitivos o de referencias a objetos.

```
void intercambio(int x, int y){
  int auxiliar; // Variable local.
  auxiliar = y;
  y = x;
  x = auxiliar;
}
```

Un método puede usar variables locales y aquellas definidas fuera del método. Puede haber coincidencia de nombres. ¿Cómo distinguirlas?.
→ *La respuesta viene dada por la duración y alcance de las variables.*

Duración y alcance de las variables

- La duración y reglas de alcance se aplican a todos los identificadores de un programa Java y definen dónde y cómo pueden usarse los identificadores.
- La duración (tiempo de vida) de un identificador es el tiempo durante el cual la información está accesible.
- El alcance (ámbito) de un identificador determina qué parte del programa ve dicho identificador.

Variables de instancia y automáticas

Variables de instancia o de ejemplar:

- Se definen en una clase, no en un método.
- Existen desde que se carga la clase en memoria hasta el final del programa.
- Si no tienen valores iniciales asignados el compilador los asigna por defecto (a 0 y **false**).

Variables automáticas:

- Se crean cuando se entra en el bloque (porción de código entre llaves) en el que se declaran .
- Existen desde que se declaran y mientras se está en el bloque en el que se declaran.
- Deben inicializarse explícitamente antes de ser usadas.
- Los parámetros y variables locales a un método son variables automáticas.

```
class Recorrido_Array {
  int a[][] = { {1,2}, {4} };
  public int suma_array( ) {
    int total = 0;
    for (int i = 0; i < a.length;  i++){
       System.out.println(i);
       System.out.println(j); // error.
       for (int j = 0; j < a[i].length; j++)
            total += a[i][j];
    }
    System.out.println(i); // error.
    return total;
  }
}
```

Es un error de sintaxis definir una variable local con el mismo nombre que uno de los argumentos del método.

En Java los bloques pueden anidarse, pero los métodos no. (No es posible definir un método dentro de otro).

La referencia this.

Las variables locales pueden recibir el mismo nombre que las variables de instancia.

Cuando una variable local se llama igual que una de instancia, oculta a la de instancia dentro de su ámbito.

Para referirnos a la variable de instancia debemos calificar al nombre de la variable con **this**.

No obstante, no es, en general, una buena práctica dar los mismos nombres a variables de instancia y variables locales.

Lo mismo ocurre si consideramos bloques. Las variables más internas ocultan a las más externas que tengan el mismo nombre.

```
class Recorrido_Array {
  int total;
  public void suma_array(int a[]) {
    int total = 0;
    for (int i = 0; i < a.length;  i++)
       total += a[i];
    this.total += total;
  }
}
```

Mejor así

```
class Recorrido_Array {
  int total;
  public void suma_array(int a[]) {
    int subtotal = 0;
    for (int i = 0; i < a.length;  i++)
       subtotal += a[i][j];
    total += subtotal;
  }
}
```

Recursión o Recursividad

La recursión es una técnica de programación mediante la cual los procedimientos y funciones se llaman a sí mismos.

→ La recursión se utiliza cuando es difícil encontrar o implementar una solución basada en bucles.

→ Ciertos problemas tienen una solución más fácil y elegante cuando en su solución se emplea un método que se llama a sí mismo.

Por ejemplo, el cálculo del número factorial de un número entero **n**:

0! = 1, por definición.

n! = n * (n-1)!

```
// Definición recursiva del método
// factorial.
public long factorial(long numero){
   if(numero == 0) return 1;
   else
      return numero * factorial(numero-1);
}
```


Recursión o Recursividad

La recursión puede emplearse si:

1. Existe un caso base que tiene solución no recursiva (el caso del cero).

   0! = 1, por definición.

   Si el método recursivo ha llegado al caso base devuelve un resultado:

   ```
   if(numero == 0) return 1;
   ```

2. Si no se ha llegado al caso base, el método se llama a sí mismo, pero considerando un problema más pequeño (**n!= n * (n-1)!**):

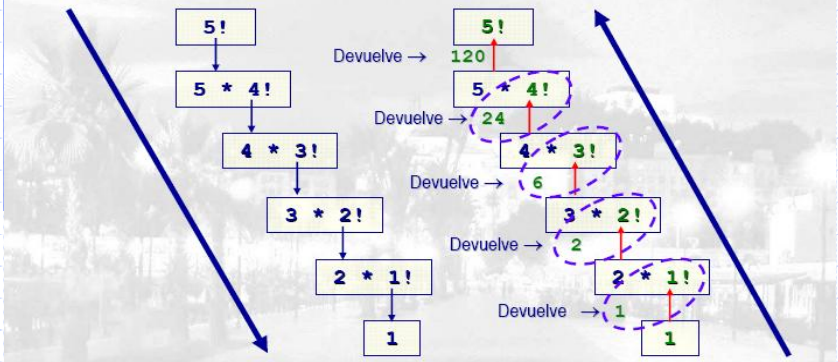
   ```
   else return numero * factorial(numero-1);
   ```

   La parte del problema que no se sabe resolver debe ser más pequeña que la actual:

   n - 1 es más pequeño que **n**.

3. Los problemas pequeños en los que se va diviendo el más grande deben *converger* (como las series matemáticas) al caso base, si no, será infinita su ejecución.

Recursión o Recursividad



La devolución del caso base desencadena una devolución de valores de manera ascendente hasta la primera ejecución del método.

Recursión: serie de *Fibonacci*

La *serie de Fibonacci* comienza en `0` y `1` y tiene la propiedad de que cada número de Fibonacci es la suma de los dos números de Fibonacci previos.

Cada llamada recursiva particiona el problema de tamaño `n` en dos de tamaño menor que convergen al caso base pues `n-1`, `n-2`, `n-3`, ... convergen a `1` y a `0`.

casos base:

```
Fibonacci(0) = 0    Fibonacci(1) = 1
```

División del problema

```
Fibonacci(n) =
      Fibonacci(n-1) + Fibonacci(n-2)
```

En Java:

```
long fibonacci(long n){
   if(n == 0 || n == 1) return n;
   else
      return fibonacci(n-1) +
             fibonacci(n-2);
}
```

Recursión e iteración

- Tanto recursión como iteración hacen uso de estructuras de control.
  - → La recursión usa el condicional.
  - → La iteración usa un bucle.
- Ambas requieren una condición de terminación.
- Ambas se aproximan gradualmente a la terminación.
  - → La iteración conforme se acerca al cumplimiento de una condición.
  - → La recursión conforme se divide el problema en otros más pequeños.
- Ambas pueden tener (por error) una ejecución potencialmente infinita.
- Si existe versión recursiva entonces existe versión iterativa.

Recursión: ventajas e inconvenientes

- La recursión requiere más memoria:
  - → Calcular fibonacci(20) requiere 21.891 llamadas.
  - → Calcular fibonacci(30) requiere 2.692.537 llamadas.
- La recursión es costosa en tiempo y recursos, pero...
  - → Refleja de manera más natural la solución a un problema lo que hace que sea más fácil de depurar y entender, además...
  - → La solución iterativa no siempre es inmediata.

Cada vez que se invoca un método se guardan en la pila:

- Las variables locales del procedimiento o función.
- Copias locales de los parámetros.
- La dirección de retorno.