Unidad 6

MÉTODOS EN JAVA

Objetivos

- Aprender a diseñar **funciones** y **procedimientos** en Java.
- Conocer el mecanismo de **paso de parámetros** en Java.
- Establecer la cabecera o interfaz de un método en Java.
- Construir correctamente las llamadas a métodos.
- Construir **resguardos y conductores** para la prueba y depuración de funciones y procedimientos.
- Diseñar **algoritmos recursivos**.
- Utilizar la traza para comprender el flujo de control.

Contenidos

- INTRODUCCIÓN
- MÉTODOS EN Java
  - Definición de métodos
  - Llamada a un método
- SENTENCIA RETURN
  - Vuelta de un método
- FUNCIONES Y PROCEDIMIENTOS
- ÁMBITO Y TIEMPO DE VIDA
- ARGUMENTOS DE LOS MÉTODOS
  - Paso por valor
  - Paso por referencia
- CÓMO DOCUMENTAR UNA FUNCIÓN
- RESGUARDOS Y CONDUCTORES
- RECURSIVIDAD

Métodos en Java

- En Java todos los subprogramas están diseñados como funciones que suelen llamarse ***métodos***.
- Existen dos puntos a considerar:
  - Definición:
    - Especifica las instrucciones que se ejecutan cuando el método es llamado.
  - Llamada:
    - Establece el flujo de control saltando a la definición del método.

En java no existe la declaración del método

Definición

//Cabecera

[Especificador]tipoDevuelto nombreMetodo (declaración-parámetros-formales)

{

<declaración variables locales> ***//Cuerpo***

<sentencias ó proposiciones>

}

```
//definición de dividir
float dividir (int a, float b) //Cabecera
{
    float div;        //Cuerpo
    div = a / b;
    return (div);
}
```

Cabecera

[Esp] tipoDevuelto nombreMetodo (Decl_parametros_Formales)

- **Especificadores de acceso:** *(De momento)*
  - **public**: accesibles desde el exterior. Es el tipo por defecto.
  - **static**: accesible sin tener que instanciar objeto alguno.
- **TipoDevuelto:** tipo de dato asociado al nombre del método.
- **nombreMetodo**: autodocumentado**.**
- **Declaración_parametros_formales**: declaración de los parámetros **formales**, separados por coma. Si no hay **no se pone nada** entre los paréntesis.

EJEMPLOS:
public static float *dividir* (int dividendo, float divisor)
// paso por valor y devolución de real asociado a *dividir*
public static void *menu* ()
// sin valor asociado a *menu*. Sin parámetros.

Llamadas a métodos

nombreFuncion ([lista-parámetros-actuales])

- Ejemplos: ¿son válidas todas las llamadas?
  - System.out.print("tengo "+edad+" años");
  - res = dividir (x, y); resultado=potencia(base,expo);
  - potencia (base, expo)= resultado;
  - System.out.print("La potencia es :"+ potencia(base, expo));
  - resultado = auxiliar + potencia (2,3);
  - res = menu(); Menu();
  - if (potencia(bas,expo)== 8)

Ejemplo

```
/**PruebaDividir.java
*/
public class PruebaDividir
{
    //Método de la clase PruebaDividir
    public static double dividir(int num, double den) //Cabecera
    {
       double div;        //Cuerpo
       div = num / den;
       return (div);
    }
    public static void main (String [] args) //Método Principal de PruebaDividir
    {
      double res;
       res= dividir(8,2.0); //Llamada a método de la clase PruebaDividir
      System.out.println("El resultado es: "+res);  //llamada a método de Java
    }
}
```

Vuelta de un método

- A través de la sentencia ***return.***
  - Formato **return [(expresión)];**
    - Es obligatoria para los métodos de tipo función
    - Es opcional para métodos de tipo procedimiento, es decir, los que su **tipoDevuelto** sea **void.**
    - Por razones de calidad, las funciones deben tener una única sentencia **return,** aunque Java permite que puedan tener más.
- Cuando se encuentra la llave de cierre.

Ver ejemplos pág.6-8

Función / Procedimiento

	FUNCIONES	PROCEDIMIENTOS
Llamada:	**VAR = nombre (parámetros reales);**	**nombre (parámetros reales);**
Cabecera:	**tipo nombre (decl_parám_formales)**	**void nombre (decl_parám_formales)**
return	**Sí.**	**No necesario.**

Llamada: MAX=**max** (A,B);
Cabecera: **int** max(int a,int b)
{
int c;
if (a > b)
c = a;
else
c = b;
Finalización: **return**(c);
}

Llamada: **error** (z);
Cabecera: **void** error (int x)
{ switch (x)
{
case 1:
System.out.print("Error, valor incorrecto")
break;
case 2:
System.out.print("Error fichero inexistente")
break;
}
}//Finalización

Paso de parámetros

- El paso de parámetros en Java se realiza **siempre por valor (¡OJO!) de una dirección de memoria**:
  - Los tipos primitivos son inmutables (no pueden ser modificados por el subprograma). Es decir, su paso es auténticamente por valor.

    Algún otro objeto también lo es, por ejemplo los declarados final.

    Por ejemplo String, declarada como public final class **String**.
  - Los objetos de clases pueden variar sus campos a través de métodos, pero no se puede cambiar un objeto. Lo que se pasa como parámetro es una copia del nombre del objeto que realmente es la dirección de comienzo en memoria de la memoria privada del objeto. Esto es en realidad exactamente como el paso por dirección de C++.
- **Paso por referencia ????, es habitual llamarle así al paso por valor de una dirección de memoria en los entornos Java, pero es erróneo.**

Ver ejemplos pág.11

Paso por valor

```
Public static void intercambiar (int parFor1, int parFor2)    //cabecera
{     int aux;
      aux = parFor1;
      parFor1 = parFor2;
      parFor2 = aux;
} //Hacer la traza del subprograma
```

Interfaz incorrecta
Proceso: Intercambia dos números
Precondiciones: ninguna
Entrada: dos números enteros (Entradas/salida : debería tener dos números enteros)
Salida: ninguna
Postcondiciones: ninguna

```
.....
.....
… void main (…)
{     int num1, num2;
      …..
      if (num1>num2)
      {
        intercambiar (num1, num2); //llamada
        System.out.print("\nLos números intercambiados    son:"+num1 " "num2);
      }
} //Hacer la traza del programa
```

//Hacer la traza integrada

El PP no se entera del intercambio porque aunque el subprograma está "bien diseñado", el cambio se produce localmente en el subprograma intercambiar .

Este problema no puede ser solucionado pues Java no permite el paso por referencia y los tipos primitivos son inmutables.

Ámbito -Tiempo de Vida

Definición: Ámbito es la parte del programa en que existe y por tanto se conoce y puede accederse un ítem de dato o de sentencia. El ámbito delimita el tiempo de vida.

- En Java, el código de un método es privado, no interactúa con el de ningún otro método. Su acceso depende del modificador con el que se definen.
- Las variables:
  - De clase: static. Compartida por todas las instancias de una clase.
  - De instancia: abarca a todos los métodos no estáticos de la clase y su acceso depende del modificador con que se las define
  - locales se declaran dentro de un método y su tiempo de vida es mientras se ejecute el método. (de momento sólo usamos estas)
- Los parámetros **se comportan como** variables locales de los métodos.
- En Java no existen variables globales.

Documentar un método

- Cada método debe tener una documentación externa con:
  - Librería donde se encuentra, o árbol jerárquico de clases en su caso.
  - Signatura.
  - Descripción: Breve información sobre su uso (interfaz), relaciones con otras funciones.
  - Ejemplo de uso.
- Documentación interna: interfaz junto al código de definición.

Técnicas de apoyo al desarrollo

Resguardo

Ver Ejemplos pag. 14

- Subprograma vacío con el mismo nombre e interfaz que la función o procedimiento que simula.

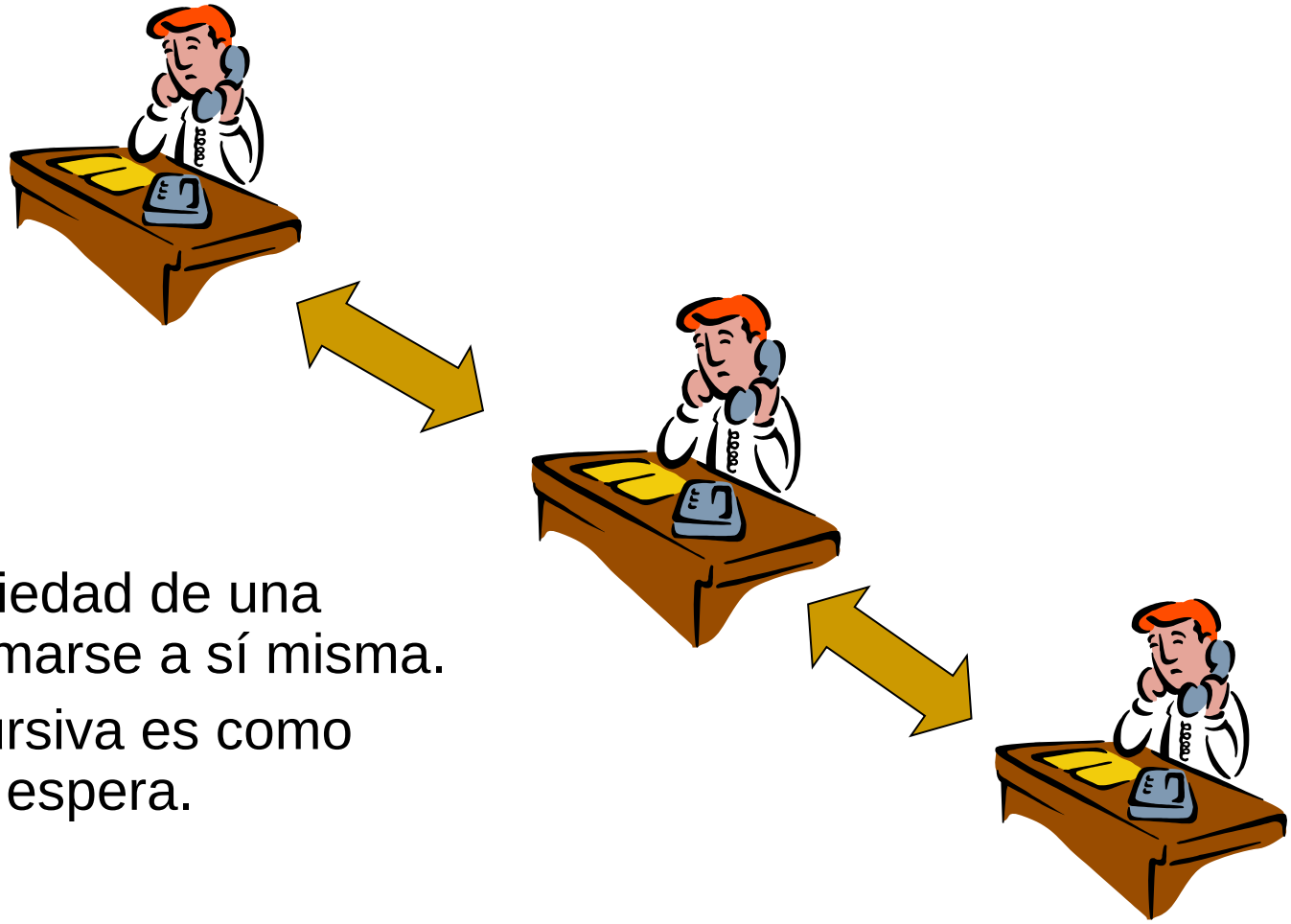
Ver Ejemplos pag. 15-17

Conductor

- Programa principal sencillo que contiene un código mínimo, con las definiciones suficientes para realizar las llamadas a procedimientos y funciones que se desean verificar.

Ambos sirven para depurar programas y probar y verificar métodos.

Recursividad



Definición: propiedad de una función para llamarse a sí misma.

La llamada recursiva es como una llamada en espera.

Recursividad

- ¿En qué consiste la recursividad?

  En el cuerpo de sentencias del subprograma se llama al propio subprograma para resolver “una versión más pequeña” del problema original.

- Aspecto de un subprograma recursivo.

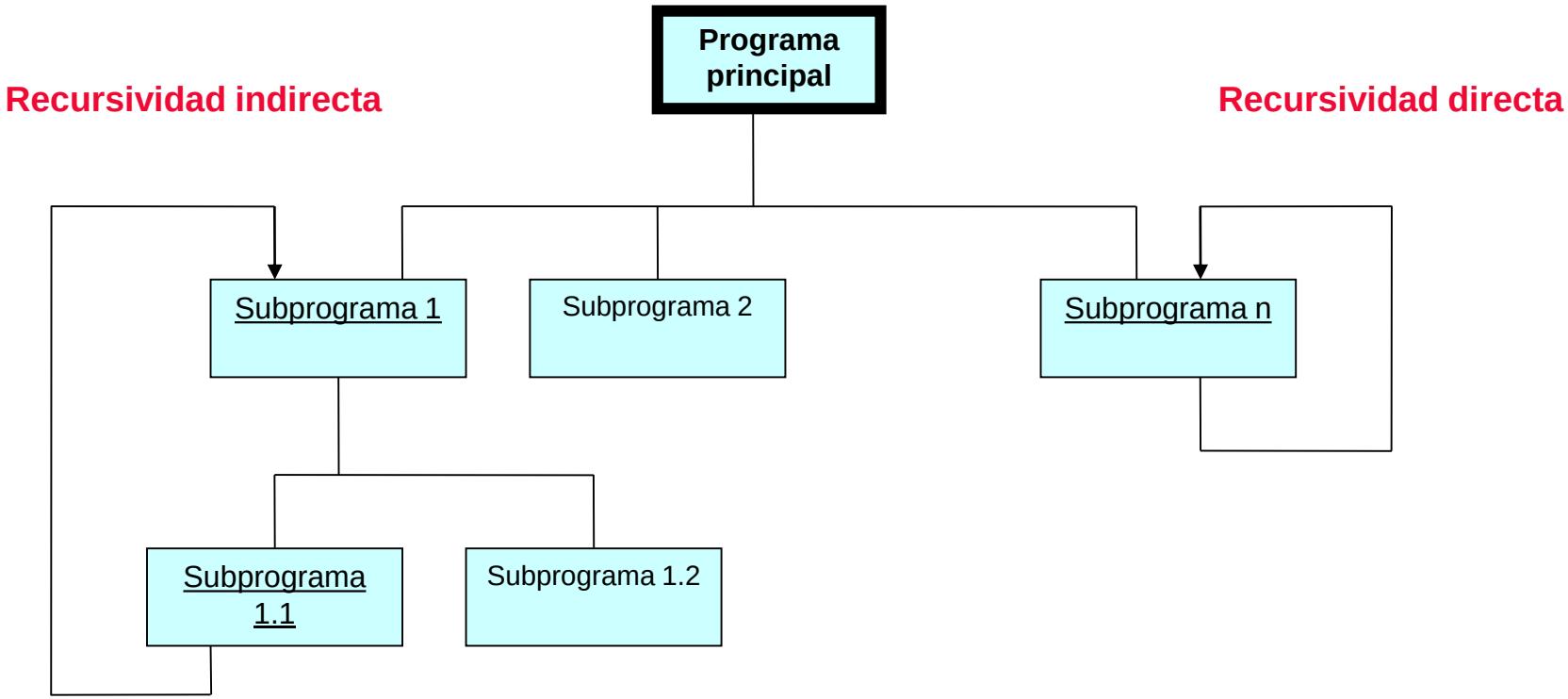
```
supRecursivo (...)
Inicio
    ...
    supRecursivo(...)
    ...
FinSR.
```

Recursividad

- La recursividad es una alternativa a la iteración.
- No todos los lenguajes admiten recursividad
- Normalmente las soluciones iterativas son más rápidas que las recursivas
- Ámbito de Aplicación:
  - General.
  - Problemas cuya solución se construye en base a la solución del mismo problema, pero con un caso de menor tamaño.
- Razones de uso:
  - En problemas cuya solución es más fácil recursiva que iterativamente (ordenación rápida Quicksort).
  - Soluciones elegantes.

Tipos de Recursividad

- **Indirecta o cruzada:** subprograma A que llama a otro subprograma B que vuelve a llamar a A.
- **Directa**: Subprograma que se llama a sí mismo.



Tipos de Recursividad

- **Recursividad simple**

  Aquella en cuya definición sólo aparece una llamada recursiva. Se puede transformar con facilidad en algoritmos iterativos.

- **Recursividad múltiple**

  Se da cuando hay más de una llamada a sí misma dentro del cuerpo de la función, resultando más difícil de hacer de forma iterativa. Un ejemplo típico es la función de Fibonacci

- **Recursividad anidada**

  En algunos de los argumentos de la llamada recursiva hay una nueva llamada a sí misma. Ejemplo típico La función de Ackermann.

Requerimientos para el diseño de programas recursivos

¡¡Ojo con la recursividad infinita!!

- Debe darse una definición recursiva del proceso, que se solucionará en función de versiones menores de sí mismo, esto se conoce como **caso general** (puede no ser único).

- Debe existir un **caso base** (puede no ser único)**,** que es aquél para el que la solución se establece de forma no recursiva. Se soluciona, generalmente, con una sentencia condicional (*if* o *switch*) .

¡¡Ojo con *Stack overflow*!!

Verificación de subprogramas recursivos

- Método inductivo de las tres preguntas:
  - **La pregunta Caso-Base**: ¿Existe al menos una salida no recursiva o **caso base** del subprograma? Además, ¿el subprograma funciona correctamente para este caso?.
  - **La pregunta Más-pequeño**: ¿Cada llamada recursiva se refiere a un caso más pequeño del problema original?.¿No hay recursión infinita?.
  - **La pregunta Caso-General**: ¿es correcta la solución en aquellos casos no base?

Recursividad. Ejemplo

Solución iterativa de factorial: **fact n= n*(n-1)*(n-2)*. . .*1**

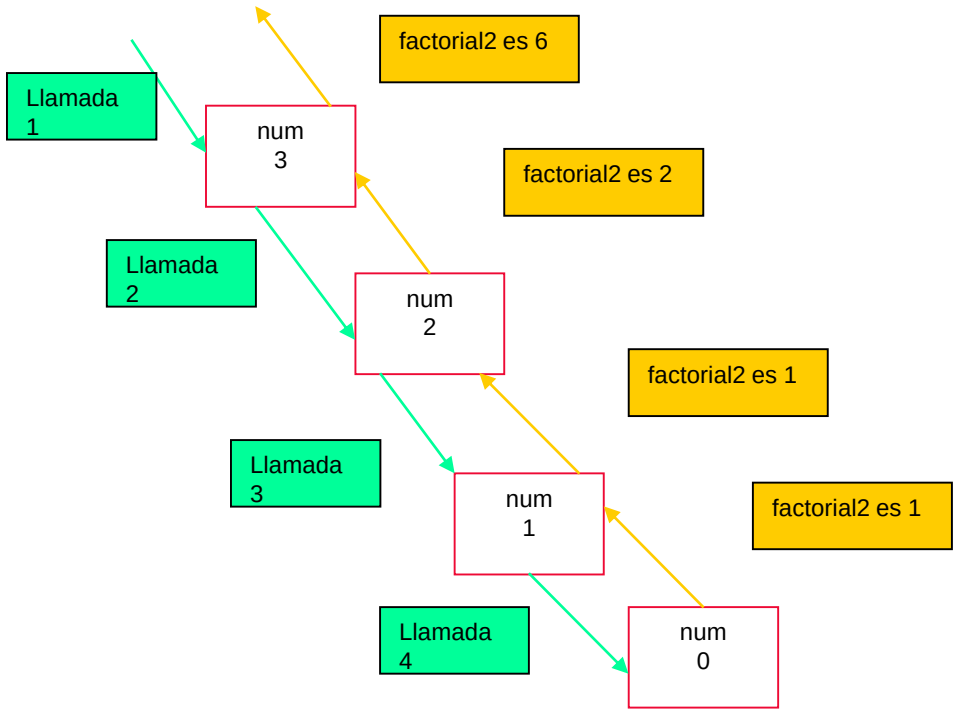
```
int factorial1(int num)
{   int solucion=1,cont;
    for (cont=1 ;cont <= num; cont++)
        solucion = solucion * cont;
    return  (solucion);
}
```

Solución recursiva de factorial

Caso general: n! = n (n-1)!

Caso base: 0! = 1

```
int factorial2 (int num)
{
     int solucion;
    if (num==0)
        solucion=1;
    else
        solucion = num * factorial2 (num-1);
    return (solucion);
}
```



Depuración

- ERRORES COMUNES .
  - Tendencia a usar estructuras iterativas en lugar de estructuras selectivas. El algoritmo no se detiene.

Comprobar el uso de SI o SEGUN

  - Ausencia de caso base.
  - Solución al problema incorrecta

Seguir el método de las 3 preguntas

¿Recursión o iteración?

- Se puede utilizar la recursividad como una alternativa a la iteración.
- Una solución recursiva es normalmente menos eficiente en términos de tiempo de ejecución que una solución iterativa, debido a las operaciones auxiliares que conllevan las llamadas adicionales a las funciones.
- La solución recursiva muchas veces permite a los programadores especificar soluciones naturales, que en otros casos sería difícil de resolver.

¿Recursión o iteración?

- Ventajas de la Recursión
  - Soluciones simples, claras.
  - Soluciones elegantes.
  - Soluciones a problemas complejos.
- Desventajas de la Recursión: ¿EFICIENCIA?

¿Recursión o iteración?

Coste informático:

- Aumento del tiempo de ejecución.
- Aumento de la memoria utilizada.
- Sobrecarga (colapso) de la Pila.

- Desventajas de la Recursión: ¿EFICIENCIA?
  - Aumento del coste informático asociado con las llamadas a funciones.
    - Una simple llamada puede generar un gran número de llamadas recursivas. (**Factorial(n)** genera **n** llamadas recursivas).
    - ¿La claridad compensa el aumento del coste?
    - El valor de la recursividad reside en el hecho de que se puede usar para resolver problemas sin fácil solución iterativa.
  - La ineficiencia inherente de algunos algoritmos recursivos.

¿Recursión o iteración?

- La recursividad se debe usar cuando no exista una solución iterativa simple.
- Cuando las dos soluciones son igualmente fáciles de implementar, se debe usar siempre la solución iterativa.
- Evitar la utilización de la recursividad en situaciones de rendimiento crítico o exigencia en tiempo y memoria, ya que las llamadas recursivas emplean mucho tiempo y consumen memoria adicional.

"Cuando la decisión de triunfar es lo suficientemente fuerte, el fracaso, jamás te alcanzará".

Anónimo