Recursividad en Java

La recursividad es una técnica potente de programación que puede utilizarse en lugar de la iteración para resolver determinados tipos de problemas.

Por ejemplo, para escribir un método que calcule el factorial de un número entero no negativo, podemos hacerlo a partir de la definición de factorial:

```
Si n = 0 entonces
0! = 1

si n>0 entonces
n! = n * (n-1) * (n-2) * ... * 3 * 2 * 1
```

Esto dará lugar a una solución iterativa en Java mediante un bucle for:

```
// Método Java no recursivo para calcular el factorial de un número
public double factorial(int n){
  double fact=1;
  int i;
  if (n==0)
    fact=1;
  else
    for(i=1;i<=n;i++)
      fact=fact*i;
 return fact;
}
```

Pero existe otra definición de factorial en función de sí misma:

$0! = 1$

$n! = n \cdot (n-1)!$, si $n > 0$ (El factorial de n es n por el factorial de n-1)

Esta definición da lugar a una solución recursiva del factorial en Java:

```
// Método Java recursivo para calcular el factorial de un número
public double factorial(int n){
  if (n==0)
    return 1;
  else
    return n*(factorial(n-1));
}
```

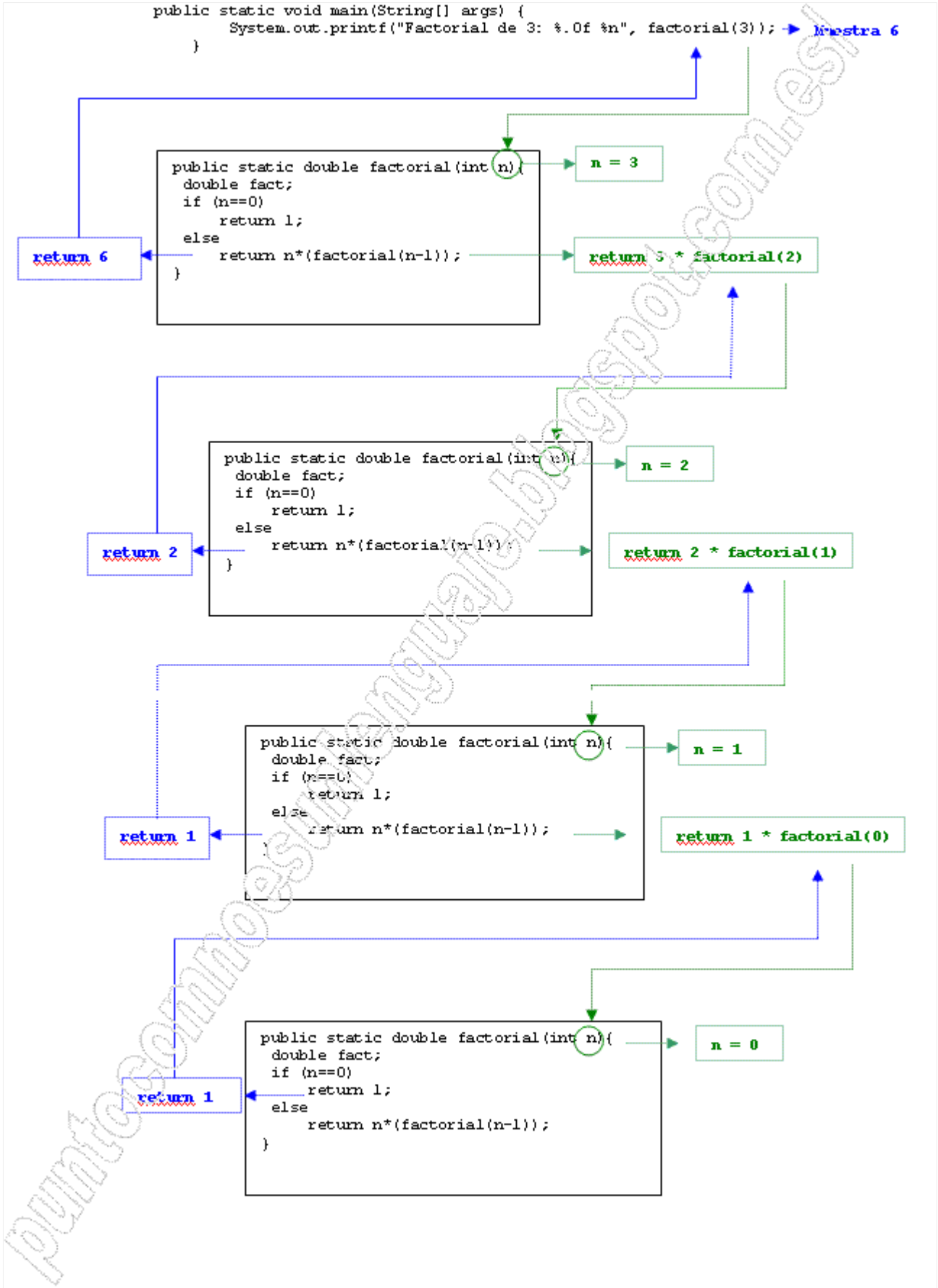
Un método es recursivo cuando entre sus instrucciones se encuentra una llamada a sí mismo.

La solución iterativa es fácil de entender. Utiliza una variable para acumular los productos y obtener la solución. En la solución recursiva se realizan llamadas al propio método con valores de n cada vez más pequeños para resolver el problema.

Cada vez que se produce una nueva llamada al método se crean en memoria de nuevo las variables y comienza la ejecución del nuevo método.

Para entender el funcionamiento de la recursividad, podemos pensar que cada llamada supone hacerlo a un método diferente, copia del original, que se ejecuta y devuelve el resultado a quien lo llamó.

En la figura siguiente podemos ver como sería la ejecución del programa Java anterior para calcular el factorial de 3.


public static void main(String[] args) {
System.out.printf("Factorial de 3: %.0f %n", factorial(3));
}
Muestra 6
public static double factorial(int n){
double fact;
if (n==0)
return 1;
else
return n*(factorial(n-1));
}
n = 3
return 6
return 3 * factorial(2)
public static double factorial(int n){
double fact;
if (n==0)
return 1;
else
return n*(factorial(n-1));
}
n = 2
return 2
return 2 * factorial(1)
public static double factorial(int n){
double fact;
if (n==0)
return 1;
else
return n*(factorial(n-1));
}
n = 1
return 1
return 1 * factorial(0)
public static double factorial(int n){
double fact;
if (n==0)
return 1;
else
return n*(factorial(n-1));
}
n = 0
return 1

Un método recursivo debe contener:

- Uno o más casos base: casos para los que existe una solución directa.
- Una o más llamadas recursivas: casos en los que se llama sí mismo

Caso base: Siempre ha de existir uno o más casos en los que los valores de los parámetros de entrada permitan al método devolver un resultado directo. Estos casos también se conocen como **solución trivial** del problema.

En el ejemplo del factorial el caso base es la condición:

```
if (n==0)

   return 1;
```

si n=0 el resultado directo es 1 No se produce llamada recursiva

Llamada recursiva: Si los valores de los parámetros de entrada no cumplen la condición del caso base se llama recursivamente al método. **En las llamadas recursivas el valor del parámetro en la llamada se ha de modificar de forma que se aproxime cada vez más hasta alcanzar al valor del caso base.**

En el ejemplo del factorial en cada llamada recursiva se utiliza n-1

```
return n * ( factorial(n-1) );
```

por lo que en cada llamada el valor de n se acerca más a 0 que es el caso base.

La recursividad es especialmente apropiada cuando el problema a resolver (por ejemplo calculo del factorial de un número) o la estructura de datos a procesar (por ejemplo los árboles) tienen una clara definición recursiva.

No se debe utilizar la recursión cuando la iteración ofrece una solución obvia. Cuando el problema se pueda definir mejor de una forma recursiva que iterativa lo resolveremos utilizando recursividad.

Para medir la eficacia de un algoritmo recursivo se tienen en cuenta tres factores:

- Tiempo de ejecución
- Uso de memoria

- Legibilidad y facilidad de comprensión

  Las soluciones recursivas suelen ser más lentas que las iterativas por el tiempo empleado en la gestión de las sucesivas llamadas a los métodos. Además consumen más memoria ya que se deben guardar los contextos de ejecución de cada método que se llama.

  A pesar de estos inconvenientes, en ciertos problemas, la recursividad conduce a soluciones que son mucho más fáciles de leer y comprender que su correspondiente solución iterativa. En estos casos una mayor claridad del algoritmo puede compensar el coste en tiempo y en ocupación de memoria.

  De todas maneras, numerosos problemas son difíciles de resolver con soluciones iterativas, y sólo la solución recursiva conduce a la resolución del problema (por ejemplo, Torres de Hanoi o recorrido de Árboles).

Ejemplo: Torres de Hanoi en Java

La leyenda.

En una antigua ciudad en la India, los monjes en un templo tienen que mover una pila de 64 discos sagrados de un lugar a otro.

Los discos son frágiles; sólo pueden ser cargados de uno en uno.

Un disco no debe nunca ser colocado arriba de otro más pequeño.

Además, solamente hay un lugar en el templo (aparte del lugar original y el lugar destino) suficientemente sagrado para poner una pila de discos allí y servir así de apoyo en el traslado de discos desde el origen hasta el destino.

Así que los monjes empiezan moviendo discos atrás y adelante, entre la pila original, la pila en el nuevo lugar de destino, y el lugar intermedio, siempre dejando las pilas en orden(el mayor en la base, el menor en la cima).

La leyenda dice además, que antes de que los monjes realicen el último movimiento para completar la torre en su nuevo lugar, el templo se reducirá a cenizas y el mundo se acabará.

Quizás esta leyenda tenga razón debido a la enorme cantidad de movimientos necesarios para cambiar de lugar los 64 discos

(2^64-1 = 18,446,744,073,709,551,615 movimientos).

Torres de Hanoi

Es un juego oriental que consta de tres columnas llamadas *origen, destino* y *auxiliar* y una serie de discos de distintos tamaños. Los discos están colocados de mayor a menor tamaño en la columna origen. El juego consiste en pasar todos los discos a la columna destino y dejarlos como estaban de mayor a menor. (el más grande en la base, el más pequeño arriba)

Las reglas del juego son las siguientes:

- Sólo se puede mover un disco cada vez.
- Para cambiar los discos de lugar se pueden usar las tres columnas.
- Nunca deberá quedar un disco grande sobre un disco pequeño.

El problema de las torres de Hanoi se puede resolver de forma muy sencilla usando la recursividad y la técnica divide y vencerás. Para ello basta con observar que si sólo hay un disco (caso base), entonces se lleva directamente de la varilla *origen* a la varilla *destino.* Si hay que llevar n>1 (caso general) discos desde *origen* a *destino* entonces:

Se llevan n-1 discos de la varilla *origen* a la *auxiliar*.

Se lleva un solo disco (el que queda) de la varilla *origen* a la *destino*

Se traen los n-1 discos de la varilla *auxiliar* a la *destino*.

Utilizando recursividad se obtiene una solución muy simple pero que sorprendentemente funciona:

```
import java.util.*;
public class Hanoi {

  public static void main(String[] args) {

    Scanner sc = new Scanner(System.in);

    int n;

    System.out.println("Numero de discos: ");

    n = sc.nextInt();

    Hanoi(n,1,2,3);  //1:origen  2:auxiliar 3:destino

  }
```

```
//Método Torres de Hanoi Recursivo

public static void Hanoi(int n, int origen,  int auxiliar, int destino){

 if(n==1)

 System.out.println("mover disco de " + origen + " a " + destino);

 else{

   Hanoi(n-1, origen, destino, auxiliar);

   System.out.println("mover disco de "+ origen + " a " + destino);

   Hanoi(n-1, auxiliar, origen, destino);

  }

 }

}
```

Ejercicio

Ejecuta el método anterior para distinto número de discos.
Escribe un método recursivo que calcule el **número de movimientos** necesarios para mover n discos.