

Clase adicional 4

Temas

- Iteración y recursión
- Alcance, acceso y duración de identificadores
- Herencia
- Problemas de la clase adicional
- Problemas de diseño

Iteración y recursión

El siguiente ejemplo muestra el uso de métodos iterativos y recursivos en la programación de Java. Consta de un método *main*, un método iterativo para calcular el coseno de un ángulo, un método recursivo para calcular el seno de un ángulo y un método recursivo para calcular el factorial de un número. La serie de ampliaciones del seno y el coseno de x se muestran a continuación (donde x está en radianes):

$$\sin(x) = \sum_{i=1}^n (-1)^{(i+1)} \frac{x^{(2i-1)}}{(2i-1)!} = \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots$$

$$\cos(x) = \sum_{i=1}^n (-1)^{(i+1)} \frac{x^{(2i-2)}}{(2i-2)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots$$

```
import javax.swing.*;
public class IterationRecursion {
    public static void main(String[] args)
    {
        /* se pide al usuario la introducción de un ángulo en radianes y del número de
        términos que se utilizarán en el cálculo */

        double x,sin_x,cos_x;
        int n;

        String xRadians= JOptionPane.showInputDialog("Escriba x (ángulo en radianes)");
        x= Double.parseDouble(xRadians);
        String nTerms= JOptionPane.showInputDialog("Escriba n (número de términos
        para la ampliación)");
        n= Integer.parseInt(nTerms);

        cos_x = cos_iter( x , n ) ; // llamada al método iterativo del coseno
        sin_x = sin_rec ( x , n ) ; // llamada al método recursivo del seno

        System.out.println("RESULTADOS" ); // imprime los resultados
```

```

System.out.println(" sen(" + x + ") = " + sin_x);
System.out.println(" cos(" + x + ") = " + cos_x );
System.out.println(" (Usando " + n + " términos de ampliación)" );
}

private static double cos_iter(double angle , int n)
{
    // Método iterativo para calcular el coseno
    double sum = 0.0 ;

    for(int i=1 ; i <= n ; i++)
    {
        if ((i+1)%2==0)
            sum += Math.pow(angle,(2.0*i-2.0)) /factorial(2*i-2);
        else
            sum -= Math.pow(angle,(2.0*i-2.0)) /factorial(2*i-2);
    }

    return sum ;
}

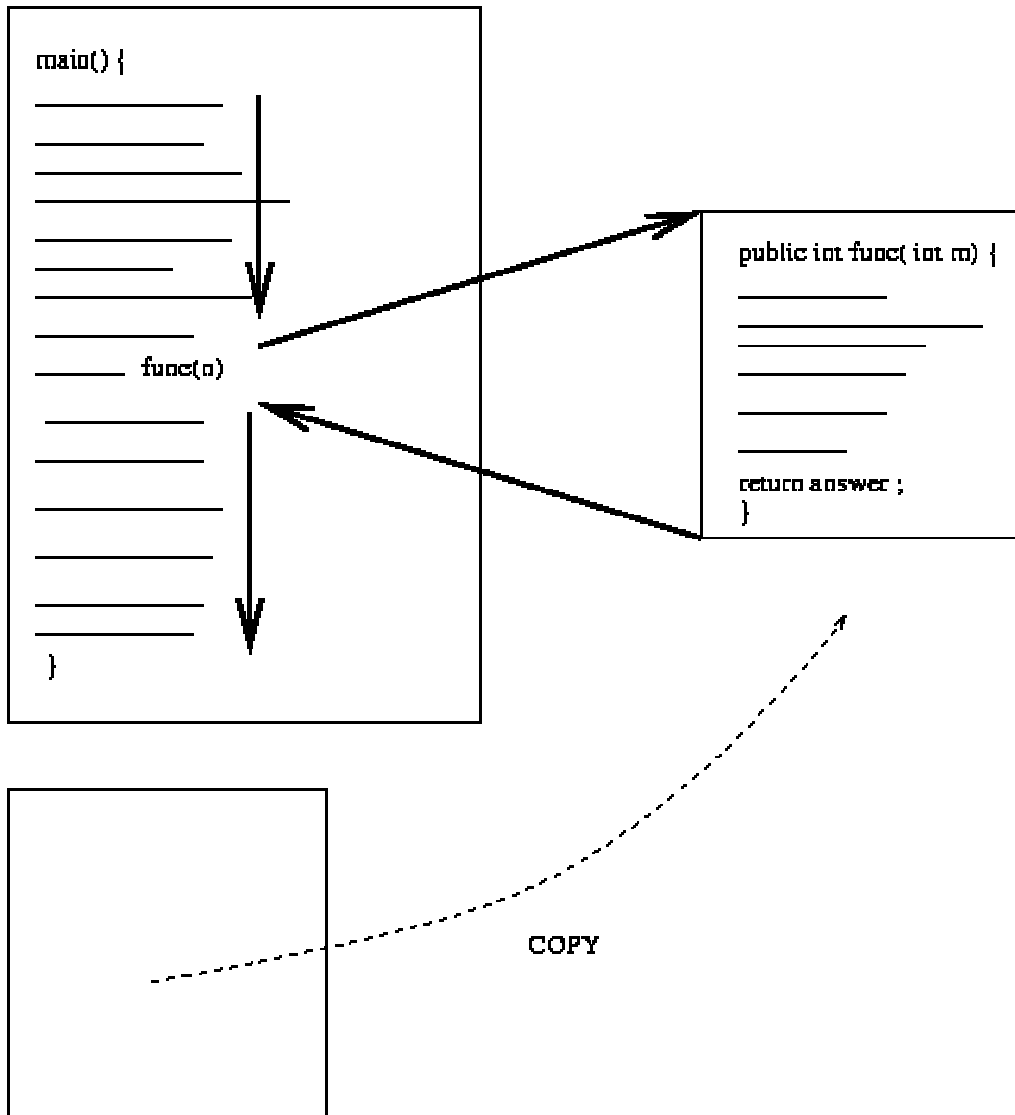
private static double sin_rec (double angle, int i)
{
    // Método recursivo para calcular el seno
    if(i==1)
        return angle;
    else if ((i+1)%2==0)
        return Math.pow(angle,2.0*i-1.0)/
            factorial(2*i-1)+sin_rec(angle,i-1);
    else
        return -Math.pow(angle,2.0*i-1.0)/
            factorial(2*i-1)+sin_rec(angle,i-1);
}

private static int factorial(int n)
{
    // calcula el factorial de n
    if(n==0)
        return 1;
    return ( n * factorial(n-1));
}
}

```

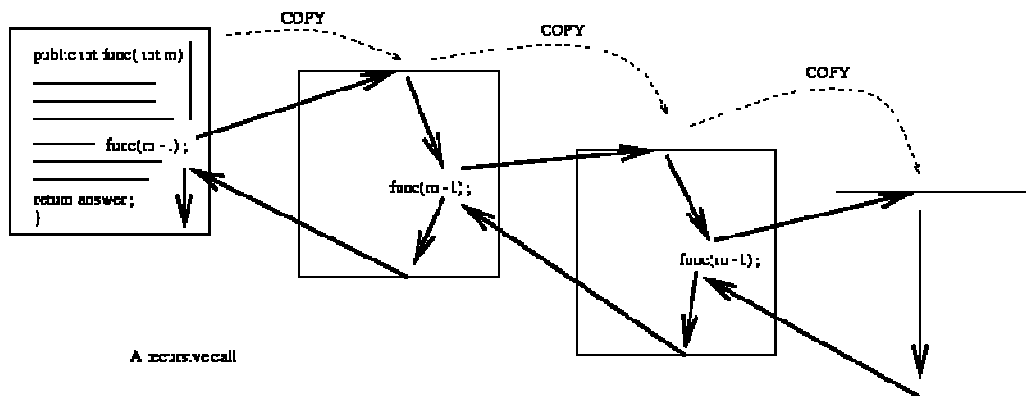
Análisis de los métodos recursivos

Se dice que un método es recursivo si se llama a sí mismo. Tal vez este concepto sea difícil de entender, por lo que requiere una explicación más detallada. En un contexto no recursivo, al llamar a un método desde otro método, lo que ocurre es lo siguiente.



Suponga que está llamando al método `func()` desde el método `main()`. El equipo primero ejecuta las sentencias del comienzo del método `main()`. Pero, al llamar al método `func()`, el equipo conceptualmente (aunque no de forma real) realiza una copia del código escrito para `func()` y comienza a ejecutarlo en esta copia con los argumentos correctos. Exista o no una sentencia *return* al final del método llamado, la llave de cierre `}` ordena al equipo a que retroceda para ejecutar el resto del método `main()`.

La siguiente figura ilustra lo que ocurre durante una llamada recursiva.



Imaginemos que el método `func()` es recursivo. Cuando este método se ejecuta, en algún punto, se llamará a sí mismo. Tal como muestra el diagrama, no hay ningún bucle involucrado, ya que se realiza una copia del método. A continuación, las sentencias de la copia comienzan a ejecutarse, aunque esta vez con distintos argumentos. Si se desencadena otra llamada recursiva, se realizará otra copia. Y así sucesivamente hasta que, en una copia del método `func()`, los argumentos permitan alcanzar el caso base y permitan que la copia haga un "return", es decir, que transfiera el control de nuevo a su padre. En ese momento, el padre toma el control, ejecuta el resto de su sentencia y, cuando llegue el momento, pasa el control a su padre. Este proceso se repite hasta que el control vuelve al método de llamada inicial, que ha estado esperando este momento todo el tiempo y ahora puede terminar.

Criterios de finalización

Dado el proceso explicado en la sección anterior, si no desea generar un número infinito de copias de un método recursivo, éste último debería tener un caso base (o varios casos) comprobados para cada entrada del método. El caso base se utiliza para determinar si la recursión debe o no finalizar. Esto es exactamente igual a las sentencias condicionales de un bucle *while* o *for* que se utilizan para finalizar la iteración. Por ejemplo, el siguiente método calcula la suma de los primeros `n` enteros positivos utilizando `n <= 1` como criterio de finalización. Observe que el criterio de finalización suele ser el primer segmento de código de un método recursivo.

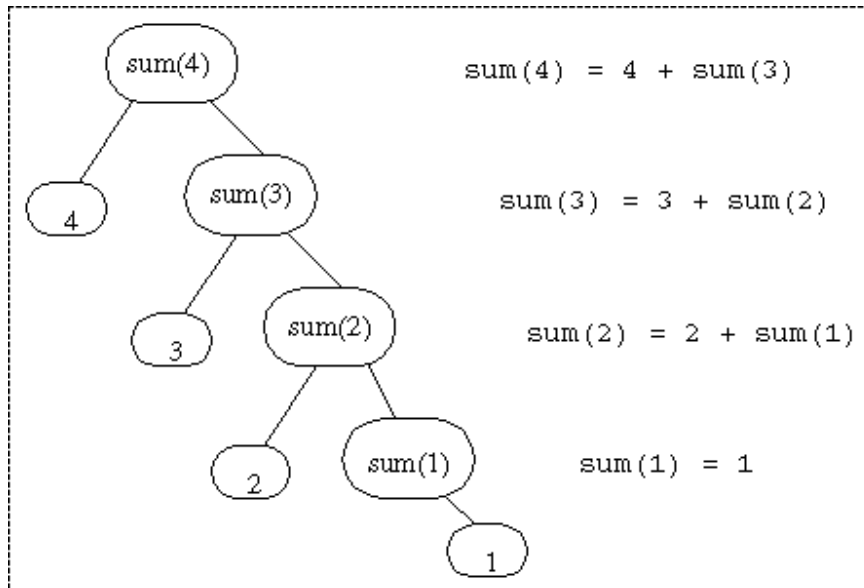
```
int sum(int n) {
if ( n<=1 )      // si n <= 1, la recursión finaliza
    return n;    // Éste es el caso base
else
    return (n+sum(n-1)); // de lo contrario, n disminuye en 1 cada vez
}                // hasta que se cumple el criterio de finalización
```

Un problema menor

La estructura típica de un método recursivo implica una llamada a sí mismo con un problema menor tras la comprobación de los criterios de finalización. Esta sentencia es necesariamente poco concisa y debe leerse siempre con precaución. En concreto, puede no resultar evidente el modo de llamar a un método con un problema menor. O incluso, cuando nos dan un método, ¿dónde se llama a sí mismo con un problema menor y qué relación existe entre el problema menor y el más grande? La esencia de la escritura de código recursivo es darse cuenta de cómo descomponer problemas en otros más pequeños que tengan las mismas características que los problemas de mayor envergadura.

Visualización de la recursión

Un modo de visualizar los métodos recursivos se basa en la utilización de un formato de árbol. Como ejemplo, para el método anterior `sum()`, `sum(4)` se puede visualizar de la siguiente forma. Puede intentar construir un árbol similar para el método factorial del último ejemplo.



Alcance, acceso y duración de identificadores

Cada identificador de un programa tiene una duración y alcance asociados (el nombre de una variable o método se denomina identificador). Las reglas de alcance y acceso determinan dónde se puede hacer referencia a un identificador en un programa.

Alcance

El alcance de una variable es el conjunto de sentencias del programa que pueden hacer referencia a dicha variable. El alcance para un identificador puede ser un alcance de clase o de bloque.

Alcance de clase

... comienza en la apertura de la llave izquierda '{' de las definiciones de la clase y finaliza en el cierre de la llave derecha '}' de la definición de la clase. Las variables de los métodos y las instancias de una clase tienen alcance de clase. Este tipo de alcance permite a los miembros de una clase invocar directamente todos los métodos definidos en la misma clase o los heredados en la clase, y también permite tener acceso directo a las variables de la instancia definidas en la clase.

Alcance de bloque

... comienza en la declaración del identificador y finaliza en la llave derecha de cierre '}' del bloque. Los parámetros del método y las variables locales (declaradas al comienzo del método y las variables declaradas en cualquier ubicación del método) tienen alcance de bloque. Cuando los bloques están anidados, no es posible que un identificador de un bloque externo tenga el mismo nombre que el de un bloque interno, pero si una variable local tiene el mismo nombre que una variable de instancia, ésta se oculta dentro del bloque.

Accesibilidad de los miembros de la clase

Java permite la existencia de especificadores de acceso de las variables de una clase y de sus métodos. Existen cuatro especificadores de acceso: *public*, *private*, *protected* y *package* (predeterminado). El especificador de acceso 'protected' se tratará más adelante. Para declarar una variable para que tenga un tipo de especificador de acceso concreto, hay que agregar el especificador de acceso a la declaración de la variable.

La tabla siguiente muestra el nivel de acceso permitido por cada especificador

Especificador	Clase	Subclase	Paquete	Mundo
private	sí	no	no	no
public	sí	sí	sí	sí
package	sí	sí	sí	no

La primera columna muestra el especificador de acceso y el resto muestran quién tiene acceso a los miembros con el identificador de acceso correspondiente. La columna Clase indica la misma clase en la que se declara el miembro. La columna Subclase indica las subclases de dicha clase. La columna Paquete representa el paquete de la clase en la que se declara la variable del miembro y Mundo indica todas las clases.

Accesibilidad de todas las clases

Las clases también pueden tener restricciones de acceso. Una clase se puede declarar como *public*, *private* o *package* (predeterminado).

- o se puede crear una instancia de las clases *public* en cualquier lugar del programa.
- o sólo se puede crear una instancia de las clases *private* a partir de sentencias del mismo archivo de código fuente.
- o se puede crear una instancia de las clases *package* a partir del cualquier código del mismo paquete

Java también permite que las clases se definan dentro de otras clases. Estas "clases interiores" tiene un alcance restringido a las clases que encierran, esto es, cualquier código de las clases internas tiene acceso a todos los miembros de la clase que encierra y viceversa. Hablaremos de esto más adelante.

Duración

La duración determina el periodo durante el cual un identificador existe en la memoria del ordenador.

Variables locales

Las variables locales tienen una duración automática. Este tipo de variables se crean cuando el control del programa entra en el bloque en el que se crearon y se destruyen cuando el control del programa sale del bloque.

Declaración *static*

Cada objeto de una clase tiene su propia copia de las variables de instancia. No obstante, a veces, cuando queremos que las clases compartan un miembro, utilizamos una declaración de miembro *static*.

Un miembro de clase *static* existe incluso si no existe ningún objeto de dicha clase. Estos miembros existen desde el momento en el que la clase que los define se carga en la memoria. Pero debe tenerse en cuenta que, incluso aunque los nombres de los miembros *static* sean válidos, es preciso tener en cuenta su alcance a la hora de utilizarlos.

Si un método se declara como *static*, se podrá llamar al método (siempre y cuando las reglas de acceso así lo permitan) utilizando el nombre de la clase. Un método *static* no puede acceder a miembros de clases no *static*. Aquí tiene un ejemplo:

```
public class WelcomeConsole {  
  
    static int count=0;  
    private int dummy=0;  
  
    public static void main(String[] args){  
  
        dummy++; // no permitido: miembro no static  
        count++;  
        System.out.println(count);  
        System.exit(0);  
    }  
}
```

El método *main* se declara como *static* para que este método se pueda llamar sin crear una instancia de esta clase y el especificador de acceso *public* aparece, ya que es posible llamar a este método *main* desde cualquier lugar (línea de comandos). Si una clase tiene un método *public static void main()* (*main* no devuelve nada) esta clase se puede ejecutar escribiendo el nombre de la máquina virtual seguido del nombre de la clase.

Declaración final

Si una variable se declara como *final*, se informa al compilador de que su valor no se cambiará nunca. Una clase de este tipo no puede ampliarse (heredar). Un método declarado como *final* no se puede ignorar en una subclase. Asimismo, los métodos *static* y *final* son, de forma implícita, *final*.

Problemas de la clase adicional

1. El funcionamiento de los métodos recursivos se basa en llamarse a sí mismos para solucionar subproblemas hasta que éstos sean lo suficientemente sencillos como para que se puedan resolver directamente. Los métodos recursivos constan de dos partes:

la parte que se encarga de los casos más simples recibe el nombre de parte base; la parte que transforma los casos más complejos se denomina parte recursiva. Para el siguiente código, identifique la parte base y la parte recursiva del método "recursivePowerOf2" y, a continuación, escriba el resultado del siguiente programa.

```
public class MyClass {
    public static void main (String argv[]) {
        System.out.print("2 elevado a la cuarta potencia es ");
        System.out.println(MyClass.recursivePowerOf2(4));
        System.out.print("2 elevado a la décima potencia es ");
        System.out.println(MyClass.recursivePowerOf2(10));
    }

    public static int recursivePowerOf2(int n) {
        if (n==0)
            return 1;
        else
            return 2*recursivePowerOf2(n-1);
    }
}
```

2. Acceso:

En el siguiente código, encuentre las cuatro sentencias no permitidas y explique el motivo.

```
package edu.mit.course.100.tutorial4.set1;
class TestA{

    private int privateData;
    public int publicData;
    int defaultData;

    public TestA(){
        privateData=0;
        publicData=1;
        defaultData=2;
    }

    private void privateMethod ( TestA a ) {
        System.out.println(privateData);
        System.out.println(a.privateDate);
    }

    private void publicMethod ( TestB b , TectC c) {
        System.out.println(b.publicData);
        System.out.println(b.defaultData);
        System.out.println(b.privateData);
        System.out.println(c.defaultData);
    }
}
```

```

}

// ----- otro package -----

package edu.mit.course.100.tutorial4.set1;
class TestB{

    private int privateData;
    public int publicData;
    int defaultData;

    public TestB () {
        privateData=0;
        publicData=1;
        defaultData=2;
    }

    private void privateMethod () {
        TestA a = new TestA();
        System.out.println(a.privateData);
    }

    private void publicMethod ( TestC c ) {
        TestA a = new TestA();
        TestB b = new TestB();
        a.publicMethod(b,c);
        a.privateMethod(a);
    }
}

// ----- y otro package -----

package edu.mit.course.100.tutorial4.set2;

class TestC{
    private int privateData;
    public int publicData;
    int defaultData;

    public TestC(){
        privateData=0;
        publicData=1;
        defaultData=2;
    }
}

```

Problemas de diseño

1. Diseñe una nueva clase *MyMath* que implemente los métodos siguientes: *sum* (suma), *factorial*, a^b (b es un entero no negativo), *sin* y *cos*. Todos estos métodos deben implementarse como métodos *static*. Escriba otra clase para probar sus métodos.

Por ejemplo, su programa deberá ser capaz de obtener los siguientes resultados:

-sum(5) es 15
-factorial(5) es 120
-power(4, 5) es 1024
-sin(5, 6) es -1.1336172989818796
-cos(5, 6) es -0.16274663800705413

Sugerencia: todos estos métodos se deben implementar como métodos recursivos o como métodos iterativos.

2. Rediseñe la clase *HeatPlate* (última diapositiva de la Clase 11)

Analice las ventajas y las desventajas de las siguientes modificaciones con su compañero:

- ¿Debería guardar *Re*, *Nu*? Si lo hace, ¿cómo puede estar seguro de que se calcularán y que el resultado estará actualizado? ¿Depende esto de la existencia de métodos *setXXX()* en la clase?
- ¿Debería guardar *hPlate*, etc.?
- Los métodos que llaman a métodos pueden resultar confusos. ¿Los cambiaría?
- Elementos de reorganización (menores):
 - Deshacerse de *retvals* innecesarios
 - ¿Renombrar los argumentos para evitar el uso de 'this'? (debatible)
- ¿Debería ser alguno de estos métodos *static*?
- Compile y pruebe/lea con el depurador