Tutorial 4

Topics

- Iteration and Recursion
- Scope, access, and duration of identifiers
- Inheritance
- Tutorial Exercises
- Design Exercises

Iteration and Recursion

The following example illustrates the use of iterative and recursive methods in Java programming. It consists of a main, an iterative method for calculating the cosine of an angle, a recursive method for calculating the sine of an angle, and a recursive method for calculating the factorial of a number. The sine and cosine series expansions of $x$ are shown below (where $x$ is in radians):

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

\[
\cos(x) = \sum_{i=1}^{n} (-1)^{i+1} \frac{x^{(2i-2)}}{(2i-2)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots
\]

```java
import javax.swing.*;
public class IterationRecursion {
    public static void main(String[] args) {
        /* prompt user for input angle in radians and number of terms to be used in the calculation */
        
        double x,sin_x,cos_x;
        int n;

        String xRadians= JOptionPane.showInputDialog("Enter x (angle in radians)");
        x= Double.parseDouble(xRadians);
        String nTerms= JOptionPane.showInputDialog("Enter n (number of terms for expansion)");
        n= Integer.parseInt(nTerms);

        cos_x = cos_iter( x , n ) ;  // call to iterative cosine method
        sin_x = sin_rec ( x , n ) ;  // call to recursive sine method

        System.out.println("RESULTS");  // print results
    }
}
```
System.out.println(" sin(" + x + ") = " + sin_x);
System.out.println(" cos(" + x + ") = " + cos_x);
System.out.println(" (Using " + n + ") expansion terms") ;
}

private static double cos_iter(double angle , int n)
{
   // Iterative method to calculate cosine
   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)
{
   // Recursive method to calculate sine
   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)
{
   // computes the factorial of n
   if(n==0)
      return 1;
   return ( n * factorial(n-1));
}

Understanding recursive methods

A method is said to be recursive if it calls itself. This concept may be difficult to grasp, and therefore needs some explanation. In a non-recursive context, when you call a method from another method, here is what happens.
Suppose you are calling the method `func()` from the `main()` method. The computer first executes the statements at the beginning of the `main()` method. But, when it reaches the call to `func()`, the computer conceptually (though not actually) makes a copy of the code you wrote for `func()`, and starts executing the code in this copy with the correct arguments. Whether or not a return statement is at the end of the called method, the closing brace `}` tells the computer to go back to the execution of the rest of the `main()` method.

The following figure illustrates what happens during a recursive call.
Let's imagine that the method func() is a recursive method. When this method executes, at some point, it calls itself. As the diagram above shows it, there is no loop involved here, since a copy of the method is made. Then, the statements in this copy start being executed, but with different arguments this time. If another recursive called is triggered, another copy is made. This goes on and on, until finally, in one copy of the method func(), the arguments allow to reach the base case, and allow the copy to “return,” i.e. to pass the control back to its parent. The parent takes control, executes the rest of its statement, and in its turn, passes the control to its parent. This goes on until the control is back to the initial calling method, which has been waiting for all this time and can now finish. Termination Criteria

Given the process explained in the previous section, if you don’t want to generate an infinite number of copies of a recursive method, the latter should have a base case (or cases) that is tested for upon entry to the method. The base case is used to determine whether the recursion should terminate. This is exactly similar to the conditional statements in a while or a for loop which are used to terminate the iteration. For example, the following method computes the sum of the first n positive integers using n <= 1 as the termination criteria. Note that the termination criteria is usually the first segment of code in a recursive method.

```java
int sum(int n) {
    if ( n<=1 )              // if n <= 1, the recursion terminates
        return n;             // This is the base case
    else
        return (n+sum(n-1));  // otherwise, n is reduced by 1 each time
}                           // until the termination criteria is met
```

A smaller problem

The typical structure of a recursive method involves a call to itself with a smaller problem after the termination criteria has been tested. This statement is necessarily vague and must be read with some caution. Particularly, it may not be obvious how to call a method with a smaller problem. Or even when you are given a method, where it is calling itself with a smaller problem and what the relation of the smaller problem to the larger problem is. The essence of writing recursive code is to realize how to decompose problems into smaller ones, which have all the characteristics of the larger problems.
Visualizing Recursion

One way of visualizing recursive methods is by using a tree format. As an example, for the above method `sum()`, `sum(4)` can be visualized as follows. You can try building a similar tree for the factorial method from the last example.

```
sum(4)       sum(4) = 4 + sum(3)
    |
    v
sum(3)       sum(3) = 3 + sum(2)
    |
    v
sum(2)       sum(2) = 2 + sum(1)
    |
    v
sum(1)       sum(1) = 1
    |
    v
1
```

Scope, access, and duration of identifiers

Each identifier in a program has associated duration and scope (The name of a variable or method is called an identifier). The scope and access rules determine where an identifier can be referred to in a program. Scope

The scope of a variable is the set of statements in the program, which can refer to that variable. The scope for an identifier can be either a class scope or block scope.

Class scope

... begins at the opening left brace '{' of the class definitions and end at the closing right brace '}' of the class definition. Methods and instance variables of a class have class scope. Class scope enables members of a class to directly invoke all methods defined in the same class or inherited into the class and also to directly access the instance variables defined in the class.

Block Scope

... begins at the identifier's declaration and ends at the terminating right brace '}' of the block. Method parameters and local variables (declared at the beginning of a method and variables declared anywhere in a method) has block scope. When blocks are nested an identifier in a outer block cannot have the same name as an identifier in an inner block, but if a local variable has the same name as an instance variable then the instance variable is hidden within the block.
Accessibility of Class members

Java allows access specifiers for members of a class variables and its methods. There are four access specifiers: public, private, protected and package (default). The access specifier ‘protected’ will be dealt with later. To declare a variable to be of a particular access specifier type, add the access specifier to the variable declaration.

The following chart shows the access level permitted by each specifier:

<table>
<thead>
<tr>
<th>Specifier</th>
<th>class</th>
<th>subclass</th>
<th>package</th>
<th>world</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>public</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>package</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

The first column shows the access specifier and the other columns show who has access to members with the corresponding access specifier. The column, class indicates the same class in which the member is declared. The column, subclass, indicates the subclasses of this class. The column, package, represents the package of the class in which the member variable is declared and world indicates all classes.

Accessibility of Classes

Classes also can have access restrictions. A class can be declared public, private or package (default).

- public classes can be instantiated anywhere in a program.
- private classes can only be instantiated from statements in the same source code file.
- package classes can be instantiated from any code in the same package

Java also allows classes to be defined inside other classes. This “inner classes” have scope restricted to their enclosing classes. i.e. any code in inner classes has access to all members of the enclosing class and vice versa. We discuss those later.

Duration

The duration determines the period during which an identifier exists in the memory of your computer.

Local variables

Local variables have automatic duration. Automatic duration variables are created when the program control enters the block in which they were created and are destroyed when the program control exits the block.
### Static Declaration

Each object of a class has its own copy of the instance variables. But sometimes, when we want classes to share a member between them, we use static member declaration.

A static class member exists even when no objects of that class exist. Static members exist from the point at which the class that define them is loaded into the memory. But it should be borne in mind that even though the names of static members are valid their use should also take into account their scope.

If a method is declared static then the method can be called (provided access rules permit this) using the class name. A static method cannot access non static class members. Here is an example:

```java
public class WelcomeConsole {

    static int count=0;
    private int dummy=0;

    public static void main(String[] args){
        dummy++; // illegal - non static member
        count++;
        System.out.println(count);
        System.exit(0);
    }
}
```

The main method is declared static so that this method can be called without instancing this class and the public access specifier is provided because this main method can be called from anywhere (command line). If a class has a public static void main( ) (main does not return anything ) method then this class can be run by typing the virtual machine name followed by the class name. Final declaration

If a variable is declared final then it tells the compiler that its value will never be changed. A class declared final cannot be extended (inherited from). A method declared final cannot be overridden in a subclass. Also static and final methods are implicitly final.

### Tutorial Exercises

1. Recursive methods work by calling themselves to solve sub problems until the sub problems are simple enough for them to solve directly. Recursive methods include
two parts: the part that handles the simplest cases is called the base part; the portion that transforms more complex cases is called the recursive part. For the following code, identify the base part and the recursive part of the method "recursivePowerOf2", then write down the result of the following program.

```java
public class MyClass {
    public static void main (String argv[]) {
        System.out.print("2 to the 4th power is ");
        System.out.println(MyClass.recursivePowerOf2(4));
        System.out.print("2 to the 10th power is ");
        System.out.println(MyClass.recursivePowerOf2(10));
    }

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

2. Access:

For the following code, please find the four statements that are illegal and explain why.

```java
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);
    }
}
```
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);
    }
}

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;
    }
}
Design Exercises

1. Design a new MyMath class that implements the following methods: sum, factorial, \( a^b \) (\( b \) is a non-negative integer), sin, and cos. All these methods should be implemented as static methods. Write another class to test your methods.

For instance, your program should be able to output the following:

- \( \text{sum}(5) \) is 15
- \( \text{factorial}(5) \) is 120
- \( \text{power}(4, 5) \) is 1024
- \( \text{sin}(5, 6) \) is -1.1336172989818796
- \( \text{cos}(5, 6) \) is -0.16274663800705413

Hint: all these methods should be implemented either as recursive methods or as iterative methods.

2. Redesign the HeatPlate class (the last slide from Lecture 11)

Discuss the pros and cons with your partner of various changes, such as:

- Should you store Re, Nu? If you do, how can you be sure they’re computed and up to date? Does this depend on whether there are setXXX() methods in the class?
- Should you store hPlate, etc?
- Methods calling methods can be confusing. Should you change that?
- Housekeeping items (minor):
  - Get rid of the stupid retvals
  - Rename arguments to avoid using ‘this’? (Debatable)
- Should any of these methods be static?
- Compile and test/read with debugger