JAVA PROGRAMMING

Week 3: More on Methods and Classes

Lecturer:

- Ho Tuan Thanh, M.Sc.

Plan

1. Controlling access to class members
2. Pass objects to methods
3. Returning objects
4. Method overloading
5. Overloading constructors
6. Recursion
7. Understanding static

Plan

1. Controlling access to class members
2. Pass objects to methods
3. Returning objects
4. Method overloading
5. Overloading constructors
6. Recursion
7. Understanding static

Java's Access Modifiers

- Three access modifiers:
  - public
  - private
  - protected
- The default access setting (in which no access modifier is used) is the same as public unless your program is broken down into packages.

Example


```
1.  class MyClass {
2.      private int alpha; // private access
3.      public int beta; // public access
4.      int gamma; // default access
5.      /* Method to access alpha. It's OK for a member of a class
6.       * to access a private member of the same class.
7.       */
8.      void setAlpha(int a) {
9.          alpha = a;
10.     }
11.     int getAlpha() {
12.         return alpha;
13.     }
14. }
```

```
1.  public class AccessDemo{
2.      public static void main(String args[]) {
3.          MyClass ob = new MyClass();
4.          /* Access to alpha is allowed only through its
5.           * accessor methods.
6.           */
7.          ob.setAlpha(-99);
8.          System.out.println("ob.alpha is " + ob.getAlpha());
9.          // You cannot access alpha like this:
10.         // ob.alpha = 10;// Wrong, alpha is private
11.         // These are OK because beta is public and
12.         // gamma is default
13.         ob.beta = 88;
14.         ob.gamma = 99;
15.     }
16. }
```

Plan

1. Controlling access to class members
2. Pass objects to methods
3. Returning objects
4. Method overloading
5. Overloading constructors
6. Recursion
7. Understanding static

PASS OBJECTS TO METHODS

- It is both correct and common to pass objects to methods in Java.

```
1.  //Objects can be passed to methods.
2.  class Block {
3.          int a, b, c;
4.          int volume;
5.          Block(int i, int j, int k) {
6.                  a = i; b = j; c = k;
7.                  volume = a * b * c;
8.          }
9.          // Return true if ob defines same block.
10.         boolean sameBlock(Block ob) {
11.                 if ((ob.a == a) & (ob.b == b) & (ob.c == c))
12.                         return true;
13.                 else
14.                         return false;
15.         }
16.         // Return true if ob has same volume.
17.         boolean sameVolume(Block ob) {
18.                 if (ob.volume == volume) return true;
19.                 else return false;
20.         }
21. }
```

```
1.  public class PassOb {
2.      public static void main(String args[]) {
3.          Block ob1 = new Block(10, 2, 5);
4.          Block ob2 = new Block(10, 2, 5);
5.          Block ob3 = new Block(4, 5, 5);
6.
7.          System.out.println("ob1 same dimensions as ob2: "
8.                                                  + ob1.sameBlock(ob2));
9.          System.out.println("ob1 same dimensions as ob3: "
10.                                                 + ob1.sameBlock(ob3));
11.         System.out.println("ob1 same volume as ob3: "
12.                                                 + ob1.sameVolume(ob3));
13.     }
14. }
```

How Arguments Are Passed

- call-by-value
  - This approach copies the value of an argument into the formal parameter of the subroutine.
  - Changes made to the parameter of the subroutine have no effect on the argument in the call.
- call-by-reference
  - A reference to an argument (not the value of the argument) is passed to the parameter.
  - Inside the subroutine, this reference is used to access the actual argument specified in the call.
  - This means that changes made to the parameter will affect the argument used to call the subroutine.

Example: Pass by value


```
1.  class Test{
2.      void noChange(int i, int j) {
3.          i = i + j;j = -j;
4.      }
5.  }
6.  public class CallByValue {
7.      public static void main(String[] args) {
8.          Test ob = new Test();
9.          int a = 15, b = 20;
10.         System.out.println("a and b before call: " + a
11.                                                 + " " + b);
12.         ob.noChange(a, b);
13.         System.out.println("a and b after call: " + a
14.                                                 + " " + b);
15.     }
16. }
```

Example: Pass by reference


```
1.  class Test1 {
2.      int a, b;
3.      Test1(int i, int j) {
4.          a = i;
5.          b = j;
6.      }
7.      /* Pass an object. Now, ob.a and ob.b in object used
8.       * in the call will be changed.
9.       */
10.     void change(Test1 ob) {
11.         ob.a = ob.a + ob.b;
12.         ob.b = -ob.b;
13.     }
14. }
```

```
1.  class PassObjRef {
2.      public static void main(String args[]) {
3.          Test1 ob = new Test1(15, 20);
4.
5.          System.out.println("ob.a and ob.b before call: " +
6.                                                  ob.a + " " + ob.b);
7.          ob.change(ob);
8.
9.          System.out.println("ob.a and ob.b after call: " +
10.                                                 ob.a + " " + ob.b);
11.     }
12. }
```

Plan

1. Controlling access to class members
2. Pass objects to methods
3. Returning objects
4. Method overloading
5. Overloading constructors
6. Recursion
7. Understanding static

RETURNING OBJECTS [1]

A method can return any type of data, including class types.

```
//Return a String object.
class ErrorMsg {
        String msgs[] = { "Output Error", "Input Error",
                          "Disk Full", "Index Out-Of-Bounds" };
    // Return the error message.
    String getErrorMsg(int i) {
        if (i >= 0 & i < msgs.length) return msgs[i];
        else return "Invalid Error Code";
    }
}
```

RETURNING OBJECTS [2]


```
1.  class ErrMsg {
2.      public static void main(String args[]) {
3.          ErrorMsg err = new ErrorMsg();
4.
5.          System.out.println(err.getErrorMsg(2));
6.          System.out.println(err.getErrorMsg(19));
7.      }
8.  }
```

```
1.  class Err {
2.      String msg; // error message
3.      int severity; //code indicating severity of error
4.
5.      Err(String m, int s) {
6.          msg = m;
7.          severity = s;
8.      }
9.  }
10. class ErrorInfo {
11.     String msgs[] = { "Output Error", "Input Error",
12.                       "Disk Full", "Index Out-Of-Bounds" };
13.     int howbad[] = { 3, 3, 2, 4 };
14.
15.     Err getErrorInfo(int i) {
16.         if (i >= 0 & i < msgs.length)
17.         return new Err(msgs[i], howbad[i]);
18.         else return new Err("Invalid Error Code", 0);
19.     }
20. }
```

```
1.  class ErrInfo {
2.      public static void main(String args[]) {
3.          ErrorInfo err = new ErrorInfo();
4.          Err e;
5.
6.          e = err.getErrorInfo(2);
7.          System.out.println(e.msg + " severity: "
8.                                                  + e.severity);
9.
10.         e = err.getErrorInfo(19);
11.         System.out.println(e.msg + " severity: "
12.                                                 + e.severity);
13.     }
14. }
```

Plan

1. Controlling access to class members
2. Pass objects to methods
3. Returning objects
4. Method overloading
5. Overloading constructors
6. Recursion
7. Understanding static

METHOD OVERLOADING

- Two or more methods within the same class can share the same name, as long as their parameter declarations are different
  - the methods are said to be overloaded, and
  - the process is referred to as method overloading.
- This is one of the ways that Java implements polymorphism.
- To overload a method: declare different versions of it.
  - The type and/or number of the parameters of each overloaded method must differ.
  - Overloaded methods may differ in their return types, too.
  - When an overloaded method is called, the version of the method whose parameters match the arguments is executed.

```
1.  class Overload {
2.      void ovlDemo() {
3.          System.out.println("No parameters");
4.      }
5.      // Overload ovlDemo for one integer parameter.
6.      void ovlDemo(int a) {
7.          System.out.println("One parameter: " + a);
8.      }
9.      // Overload ovlDemo for two integer parameters.
10.     int ovlDemo(int a, int b) {
11.         System.out.println("Two parameters: " + a + " " + b);
12.         return a + b;
13.     }
14.     // Overload ovlDemo for two double parameters.
15.     double ovlDemo(double a, double b) {
16.         System.out.println("Two double parameters: "
17.                                                 + a + " " + b);
18.         return a + b;
19.     }
20. }
```

```
1.  class OverloadDemo {
2.      public static void main(String args[]) {
3.          Overload ob = new Overload();
4.          int resI; double resD;
5.          // call all versions of ovlDemo()
6.          ob.ovlDemo();
7.          System.out.println();
8.
9.          ob.ovlDemo(2);
10.         System.out.println();
11.
12.         resI = ob.ovlDemo(4, 6);
13.         System.out.println("Result of ob.ovlDemo(4, 6): "
14.                                                         + resI);
15.         System.out.println();
16.
17.         resD = ob.ovlDemo(1.1, 2.32);
18.         System.out.println("Result of ob.ovlDemo(1.1, 2.2): "
19.                                                 + resD);
20.     }
21. }
```

- The difference in their return types is insufficient for the purposes of overloading...
- Java provides certain automatic type conversions.
  - These conversions also apply to parameters of overloaded methods.

```
/* Automatic type conversions can affect  overloaded method resolution. */
class Overload2 {
	void f(int x) {
		System.out.println("Inside f(int): " + x);
	}
	void f(double x) {
		System.out.println("Inside f(double): " + x);
	}
}
```

```
1.  class TypeConv {
2.      public static void main(String args[]) {
3.          Overload2 ob = new Overload2();
4.          int i = 10;
5.          double d = 10.1;
6.          byte b = 99;
7.          short s = 10;
8.          float f = 11.5F;
9.
10.         ob.f(i); // calls ob.f(int)
11.         ob.f(d); // calls ob.f(double)
12.         ob.f(b); // calls ob.f(int) -- type conversion
13.         ob.f(s); // calls ob.f(int) -- type conversion
14.         ob.f(f); // calls ob.f(double) -- type conversion
15.     }
16. }
```

Another version ...


```
1.  //Add f(byte).
2.  class Overload2 {
3.      void f(byte x) {
4.          System.out.println("Inside f(byte): " + x);
5.      }
6.
7.      void f(int x) {
8.          System.out.println("Inside f(int): " + x);
9.      }
10.
11.     void f(double x) {
12.         System.out.println("Inside f(double): " + x);
13.     }
14. }
```

```
1.  class TypeConv {
2.      public static void main(String args[]) {
3.          Overload2 ob = new Overload2();
4.
5.          int i = 10; double d = 10.1;
6.
7.          byte b = 99; short s = 10; float f = 11.5F;
8.
9.          ob.f(i); // calls ob.f(int)
10.         ob.f(d); // calls ob.f(double)
11.         ob.f(b); // calls ob.f(byte) --now, no type conversion
12.         ob.f(s); // calls ob.f(int) -- type conversion
13.         ob.f(f); // calls ob.f(double) -- type conversion
14.     }
15. }
```

Plan

1. Controlling access to class members
2. Pass objects to methods
3. Returning objects
4. Method overloading
5. Overloading constructors
6. Recursion
7. Understanding static

OVERLOADING CONSTRUCTORS

Like methods, constructors can also be overloaded.

```
1.  class MyClass1 {
2.      int x;
3.      MyClass1() {System.out.println("Inside MyClass().");  x = 0;
4.      }
5.      MyClass1(int i) {
6.          System.out.println("Inside MyClass(int).");  x = i;
7.      }
8.      MyClass1(double d) {
9.          System.out.println("Inside MyClass(double).");
10.         x = (int) d;
11.     }
12.     MyClass1(int i, int j) {
13.         System.out.println("Inside MyClass(int, int).");
14.         x = i * j;
15.     }
16. }
```

```
1.  class OverloadConsDemo {
2.      public static void main(String args[]) {
3.          MyClass1 t1 = new MyClass1();
4.          MyClass1 t2 = new MyClass1(88);
5.          MyClass1 t3 = new MyClass1(17.23);
6.          MyClass1 t4 = new MyClass1(2, 4);
7.
8.          System.out.println("t1.x: " + t1.x);
9.          System.out.println("t2.x: " + t2.x);
10.         System.out.println("t3.x: " + t3.x);
11.         System.out.println("t4.x: " + t4.x);
12.     }
13. }
```

```
1.  class Summation {
2.      int sum;
3.      // Construct from an int.
4.      Summation(int num) {
5.          sum = 0;
6.          for (int i = 1; i <= num; i++) sum += i;
7.      }
8.      // Construct from another object.
9.      Summation(Summation ob) { sum = ob.sum; }
10. }
11.
12. class SumDemo {
13.     public static void main(String args[]) {
14.         Summation s1 = new Summation(5);
15.         Summation s2 = new Summation(s1);
16.
17.         System.out.println("s1.sum: " + s1.sum);
18.         System.out.println("s2.sum: " + s2.sum);
19.     }
20. }
```

Plan

1. Controlling access to class members
2. Pass objects to methods
3. Returning objects
4. Method overloading
5. Overloading constructors
6. Recursion
7. Understanding static

RECURSION

- A method can call itself : recursion.
- In general: recursion is the process of defining something in terms of itself and is somewhat similar to a circular definition.
- Recursion is a powerful control mechanism.

Example


```
1.  class Factorial {
2.      // This is a recursive function.
3.      int factR(int n) {
4.          int result;
5.          if (n == 1) return 1;
6.          result = factR(n - 1) * n;
7.          return result;
8.      }
9.      // This is an iterative equivalent.
10.     int factI(int n) {
11.         int t, result = 1;
12.         for (t = 1; t <= n; t++) result *= t;
13.         return result;
14.     }
15. }
```

```
1.  class Recursion {
2.      public static void main(String args[]) {
3.          Factorial f = new Factorial();
4.
5.          System.out.println("Factorials using recursive method.");
6.          System.out.println("Factorial of 3 is " + f.factR(3));
7.          System.out.println("Factorial of 4 is " + f.factR(4));
8.          System.out.println("Factorial of 5 is " + f.factR(5));
9.          System.out.println();
10.
11.         System.out.println("Factorials using iterative method.");
12.         System.out.println("Factorial of 3 is " + f.factI(3));
13.         System.out.println("Factorial of 4 is " + f.factI(4));
14.         System.out.println("Factorial of 5 is " + f.factI(5));
15.     }
16. }
```

Plan

1. Controlling access to class members
2. Pass objects to methods
3. Returning objects
4. Method overloading
5. Overloading constructors
6. Recursion
7. Understanding static

UNDERSTANDING STATIC

- When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.
- You can declare both methods and variables to be static.
- The most common example of a static member is main().
  - main() is declared as static because it must be called by the JVM when your program begins.
- Outside the class, to use a static member, you need only specify the name of its class followed by the dot operator. No object needs to be created.

Example: static variable


```
1.  //Use a static variable.
2.  class StaticDemo {
3.      int x; // a normal instance variable
4.      static int y; // a static variable
5.
6.      // Return the sum of the instance variable x
7.      // and the static variable y.
8.      int sum() {
9.          return x + y;
10.     }
11. }
```

```
1.  class SDemo {
2.      public static void main(String args[]) {
3.          StaticDemo ob1 = new StaticDemo();
4.          StaticDemo ob2 = new StaticDemo();
5.
6.          // Each object has its own copy of an instance variable.
7.          ob1.x = 10; ob2.x = 20;
8.          System.out.println("Of course, ob1.x and ob2.x "
9.                                                  + "are independent.");
10.         System.out.println("ob1.x: " + ob1.x + "\nob2.x: "
11.                                                 + ob2.x);
12.         System.out.println();
13.
14.         // Each object shares one copy of a static variable.
15.         System.out.println("The static variable y is shared.");
16.         StaticDemo.y = 19;
17.         System.out.println("Set StaticDemo.y to 19.");
18.         ...
```

```
1.              ...
2.              System.out.println("ob1.sum(): " + ob1.sum());
3.              System.out.println("ob2.sum(): " + ob2.sum());
4.              System.out.println();
5.              StaticDemo.y = 100;
6.              System.out.println("Change StaticDemo.y to 100");
7.
8.              System.out.println("ob1.sum(): " + ob1.sum());
9.              System.out.println("ob2.sum(): " + ob2.sum());
10.             System.out.println();
11.         }
12. }
```

```
Of course, ob1.x and ob2.x are independent.
ob1.x: 10
ob2.x: 20

The static variable y is shared.
Set StaticDemo.y to 19.
ob1.sum(): 29
ob2.sum(): 39

Change StaticDemo.y to 100
ob1.sum(): 110
ob2.sum(): 120
```

Example: static method


```
1.  class StaticMeth {
2.      static int val = 1024; // a static variable
3.      // a static method
4.      static int valDiv2() { return val / 2; }
5.  }
6.  class SDemo2 {
7.      public static void main(String args[]) {
8.          System.out.println("val is " + StaticMeth.val);
9.          System.out.println("StaticMeth.valDiv2(): "
10.                                  + StaticMeth.valDiv2());
11.         StaticMeth.val = 4;
12.         System.out.println("val is " + StaticMeth.val);
13.         System.out.println("StaticMeth.valDiv2(): "
14.                                  + StaticMeth.valDiv2());
15.     }
16. }
```

Restrictions

Methods declared as `static` have several restrictions:

- They can directly call only other `static` methods in their class.
- They can directly access only `static` variables in their class.
- They do not have a `this` reference.

```
1.  class StaticError {
2.      int denom = 3; // a normal instance variable
3.      static int val = 1024; // a static variable
4.      /*
5.       * Error! Can't access a non-static variable from within
6.       * a static method.
7.       */
8.      static int valDivDenom() {
9.          return val / denom; // won't compile!
10.     }
11. }
```

Static Blocks

- Sometimes a class will require some type of initialization before it is ready to create objects.
- It also might need to initialize certain static variables before any of the class' static methods are used.
- To handle these types of situations: Java allows you to declare a static block.
- A static block is executed when the class is first loaded.
- It is executed before the class can be used for any other purpose.

Example


```
1.  class StaticBlock {
2.      static double rootOf2;
3.      static double rootOf3;
4.
5.      static {
6.          System.out.println("Inside static block.");
7.          rootOf2 = Math.sqrt(2.0);
8.          rootOf3 = Math.sqrt(3.0);
9.      }
10.
11.     StaticBlock(String msg) {
12.         System.out.println(msg);
13.     }
14. }
```

```
1.  class SDemo3 {
2.      public static void main(String args[]) {
3.          StaticBlock ob = new StaticBlock(
4.                                          "Inside Constructor");
5.
6.          System.out.println("Square root of 2 is " +
                               StaticBlock.rootOf2);
7.          System.out.println("Square root of 3 is " +
8.                             StaticBlock.rootOf3);
9.      }
10. }
```

1. Create a class Student with the following attributes: StudentID, FullName, DateOfBirth.
2. Add constructors for Student
3. Add methods to this class to get information from these attributes.
4. Display the number of instances generated from class Student.
5. Override the method toString() to display information of each student.

Create a class RandomNumber that helps:

1. Returning a random number.
2. Returning a random value within [0,range].
3. Returning a random number within [min, max].
4. Returning n different values.
5. Returning n different values within [0,range].
6. Returning n different values within [min, max].

Create a class MyPoint that has two values x and y

1. Create a class Triangle that has three points (Using class MyPoint)
2. Add constructors for Triangle
3. Return Triangle type depending on its three points.

QUESTION ?