Course: Object Based Modeling
Code: CS-33105
Branch: MCA-3

Lecture 7: Polymorphism, Overloading Methods & Recursion

Dr. J Sathish Kumar (JSK)
(Faculty & Coordinator)

Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology Allahabad,
Prayagraj-211004

Overloading Methods

- In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different.
- When this is the case, the methods are said to be overloaded, and the process is referred to as *method overloading*.
- Method overloading is one of the ways that Java supports polymorphism.
- When an overloaded method is invoked, Java uses the type and/or number of arguments
  as its guide to determine which version of the overloaded method to actually call.
  - Thus, overloaded methods must differ in the type and/or number of their parameters.
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Example #1

```
// Demonstrate method overloading.
class OverloadDemo {
  void test() {
    System.out.println("No parameters");
  }

  // Overload test for one integer parameter.
  void test(int a) {
    System.out.println("a: " + a);
  }

  // Overload test for two integer parameters.
  void test(int a, int b) {
    System.out.println("a and b: " + a + " " + b);
  }

  // Overload test for a double parameter
  double test(double a) {
    System.out.println("double a: " + a);
    return a*a;
  }
}
```

```
class Overload {
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    double result;

    // call all versions of test()
    ob.test();
    ob.test(10);
    ob.test(10, 20);
    result = ob.test(123.25);
    System.out.println("Result of ob.test(123.25): " + result);
  }
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

Overloading Methods

- When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.
- However, this match need not always be exact.
- In some cases, Java's automatic type conversions can play a role in overload resolution.

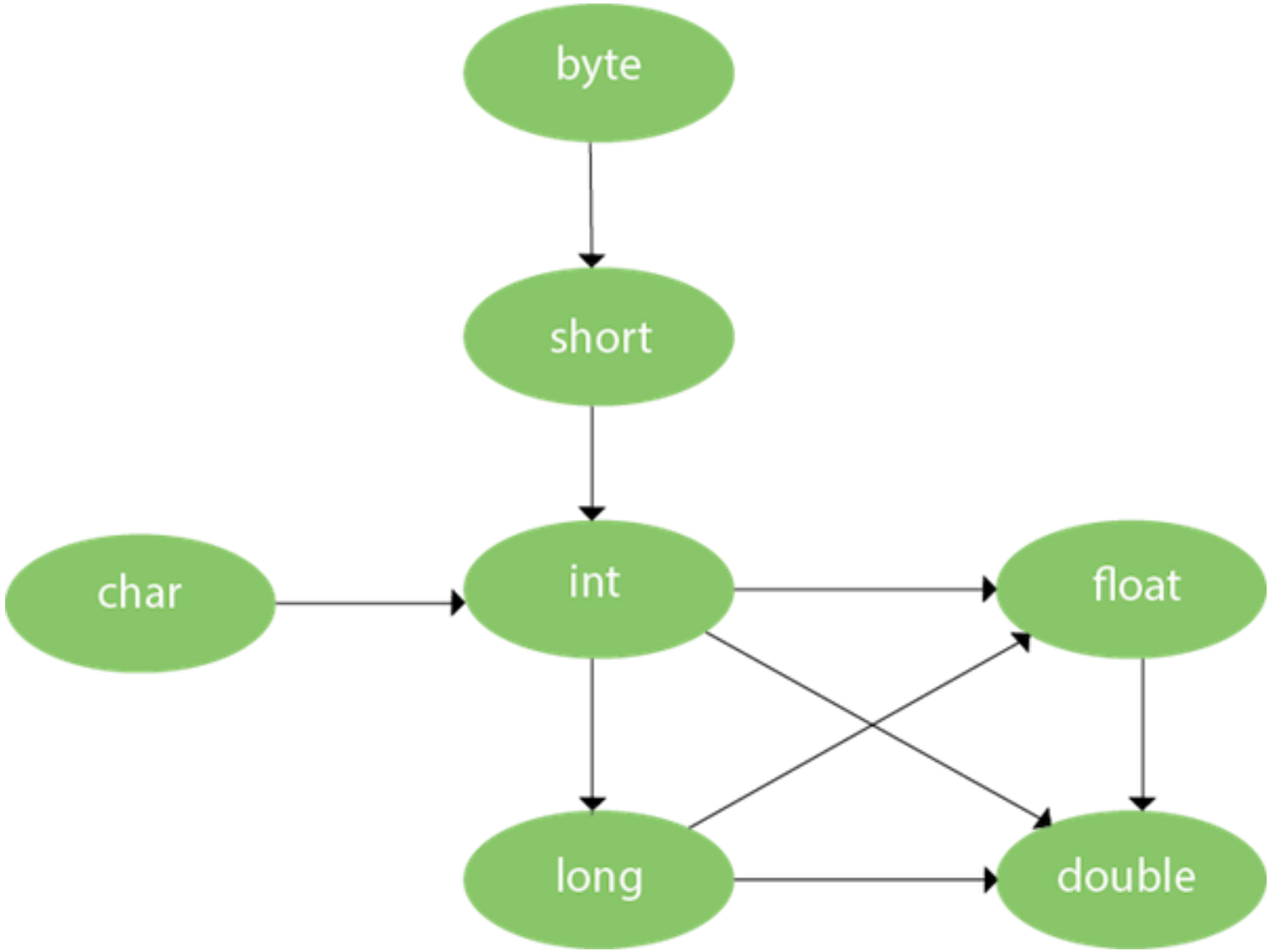
This program generates the following output:

```
No parameters
a and b: 10 20
Inside test(double) a: 88
Inside test(double) a: 123.2
```

```
// Automatic type conversions apply to overloading.
class OverloadDemo {
  void test() {
    System.out.println("No parameters");
  }

  // Overload test for two integer parameters.
  void test(int a, int b) {
    System.out.println("a and b: " + a + " " + b);

  }

  // Overload test for a double parameter
  void test(double a) {
    System.out.println("Inside test(double) a: " + a);
  }
}

class Overload {
  public static void main(String args[]) {
    OverloadDemo ob = new OverloadDemo();
    int i = 88;

    ob.test();
    ob.test(10, 20);

    ob.test(i); // this will invoke test(double)
    ob.test(123.2); // this will invoke test(double)
  }
}
```

Example #2

Method Overloading with Type Promotion



Polymorphism and Overloading Methods

- *Polymorphism* (from Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions.
- The specific action is determined by the exact nature of the situation.
- Consider a stack (which is a last-in, first-out list).
- You might have a program that requires three types of stacks.
  - One stack is used for integer values, one for floating point values, and one for characters.
- The algorithm that implements each stack is the same, even though the data being stored differs.
- In a non–object-oriented language, you would be required to create three different sets of stack routines, with each set using different names.
- However, because of polymorphism, in Java you can specify a general set of stack routines that all share the same names.
- More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.”

Constructors

- It can be tedious to initialize all of the variables in a class each time an instance is created.
- Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created.
- This automatic initialization is performed through the use of a constructor.
- A *constructor* initializes an object immediately upon creation.
- It has the same name as the class in which it resides and is syntactically similar to a method.
- It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.
- Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself

Example #3

```
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box() {
    System.out.println("Constructing Box");
    width = 10;
    height = 10;
    depth = 10;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

Both mybox1 and mybox2 were initialized by the Box( ) constructor when they were created

```
class BoxDemo6 {
  public static void main(String args[]) {
    // declare, allocate, and initialize Box objects
    Box mybox1 = new Box();
    Box mybox2 = new Box();

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
  }
}
```

When this program is run, it generates the following results:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

Overloading Constructors

Example #4

- As overloading normal methods, you can also overload constructor methods.
- **Box( )** constructor requires three parameters.
- This means that all declarations of **Box** objects must pass three arguments to the **Box( )** constructor.
- For example, the following statement is currently invalid:
  Box ob = new Box();
- Since **Box( )** requires three arguments, it's an error to call it without them.
- What if you simply wanted a box and did not care (or know)
  what its initial dimensions were?
- What if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions?

```
class Box {
  double width;
  double height;
  double depth;

  // This is the constructor for Box.
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

Example #5

```
/* Here, Box defines three constructors to initialize
   the dimensions of a box various ways.
*/
class Box {
  double width;
  double height;
  double depth;

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

```
class OverloadCons {
  public static void main(String args[]) {
    // create boxes using the various constructors
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);

     // get volume of cube
     vol = mycube.volume();
     System.out.println("Volume of mycube is " + vol);
   }
 }
```

The output produced by this program is shown here:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

Using Objects as Parameters

- The **equalTo( )** method inside **Test** compares two objects for equality and returns the result.
- That is, it compares the invoking object with the one that it is
passed.
- If they contain the same values, then the method returns **true**. Otherwise, it returns **false**.
- One of the most common uses of object parameters involves constructors.
- Frequently, you will want to construct a new object so that it is initially the same as some existing object.

Example #6

```
// Objects may be passed to methods.
class Test {
  int a, b;

  Test(int i, int j) {
    a = i;
    b = j;
  }

  // return true if o is equal to the invoking object
  boolean equalTo(Test o) {
    if(o.a == a && o.b == b) return true;
    else return false;
  }
}

class PassOb {
  public static void main(String args[]) {
    Test ob1 = new Test(100, 22);
    Test ob2 = new Test(100, 22);
    Test ob3 = new Test(-1, -1);

    System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
    System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
  }
}
```

This program generates the following output:

```
ob1 == ob2: true
ob1 == ob3: false
```

Example #7

```
// Here, Box allows one object to initialize another.

class Box {
  double width;
  double height;
  double depth;

  // Notice this constructor. It takes an object of type Box.
  Box(Box ob) { // pass object to constructor
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // constructor used when all dimensions specified
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // constructor used when no dimensions specified
  Box() {
    width = -1;  // use -1 to indicate
    height = -1; // an uninitialized
    depth = -1;  // box
  }

  // constructor used when cube is created
  Box(double len) {
    width = height = depth = len;
  }

  // compute and return volume
  double volume() {
    return width * height * depth;
  }
}
```

```
class OverloadCons2 {
  public static void main(String args[]) {
    // create boxes using the various constructors


    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);

    Box myclone = new Box(mybox1); // create copy of mybox1

    double vol;

    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);

    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);

    // get volume of cube
    vol = mycube.volume();
    System.out.println("Volume of cube is " + vol);

   // get volume of clone
    vol = myclone.volume();
    System.out.println("Volume of clone is " + vol);
  }
}
```

Call by Value

```
// Primitive types are passed by
class Test {
  void meth(int i, int j) {
    i *= 2;
    j /= 2;
  }
}
```

Example #8

```
class CallByValue {
  public static void main(String args[]) {
    Test ob = new Test();

    int a = 15, b = 20;

    System.out.println("a and b before call: " +
                       a + " " + b);

    ob.meth(a, b);

    System.out.println("a and b after call: " +
                       a + " " + b);
  }
}
```

The output from this program is shown here:

```
a and b before call: 15 20
a and b after call: 15 20
```

Call by Reference

```
// Objects are passed through their references.

class Test {
  int a, b;

  Test(int i, int j) {
    a = i;
    b = j;
  }

  // pass an object
  void meth(Test o) {
    o.a *= 2;
    o.b /= 2;
  }
}

class PassObjRef {
  public static void main(String args[]) {
    Test ob = new Test(15, 20);
    System.out.println("ob.a and ob.b before call: " +
                       ob.a + " " + ob.b);

    ob.meth(ob);

    System.out.println("ob.a and ob.b after call: " +
                       ob.a + " " + ob.b);
  }
}
```

This program generates the following output:

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```

Example #9

Recursion

- Java supports *recursion*.
- Recursion is the process of defining something in terms of itself.
- As it relates to Java programming, recursion is the attribute that allows a method to call itself.
- A method that calls itself is said to be *recursive*.

Example #10: Factorial

```
// A simple example of recursion.
class Factorial {
  // this is a recursive method
  int fact(int n) {
    int result;

    if(n==1) return 1;
    result = fact(n-1) * n;
    return result;
  }
}

class Recursion {
  public static void main(String args[]) {
    Factorial f = new Factorial();

    System.out.println("Factorial of 3 is " + f.fact(3));
    System.out.println("Factorial of 4 is " + f.fact(4));
    System.out.println("Factorial of 5 is " + f.fact(5));
  }
}
```

The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

Tutorial #3

1. Create a class to print the area of a square and a rectangle. The class has two methods with the same name but different number of parameters. The method for printing area of rectangle has two parameters which are length and breadth respectively while the other method for printing area of square has one parameter which is side of square.
2. Write a Java program using Recursion for the following
   a) Fibonacci Series
   b) Merge Sort
   c) Quick Sort
   d) Heap Sort
3. Create a class 'Student' with three data members which are name, age and address. The constructor of the class assigns default values name as "unknown", age as '0' and address as "not available". It has two members with the same name 'setInfo'. First method has two parameters for name and age and assigns the same whereas the second method takes has three parameters which are assigned to name, age and address respectively. Print the name, age and address of 10 students.
   Hint - Use array of objects