A Closer Look at Methods and Classes

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 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

- The return type alone is insufficient to distinguish two versions of a method

```
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);
}
}
```

- In some cases Java's automatic type conversions can play a role in overload resolution

- Java will employ its automatic type conversions only if no exact match is found

```
// 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();
ob.test();
ob.test(10, 20);
int i = 88;
ob.test(i); // this will invoke test(double)
ob.test(123.2); // this will invoke test(double)
}
}
```

This program generates the following output:

```
No parameters
a and b: 10 20
Inside test(double) a: 88.0
Inside test(double) a: 123.2
```

Overloading Constructors

```
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);
}
}
```

Using Objects as Parameters

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

Copy constructor

```
class Box {
  double width;
  double height;
  double depth;
  // construct clone of an object
  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;
```

Copy constructor

```
    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
```

Copy constructor

```
  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);
    double vol;
```

Copy constructor

```
        // 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);
    }
}
```

Passing arguments

- There are two ways that a computer language can pass an argument to a subroutine

- Call by value

- Call by reference

Contd..

- when you pass a simple type to a method, it is passed by value

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

Contd..

```
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
```

Contd..

- Objects are passed by reference

- Changes to the object inside the method *do* affect the object used as an argument

```
// Objects are passed by reference.
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 CallByRef {
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
```

Returning Objects

- A method can return any type of data, including class types that you create

```
// Returning an object.
class Test {
int a;
Test(int i) {
a = i;
}
Test incrByTen() {
Test temp = new Test(a+10);
return temp;
}
}
```

```
class RetOb {
public static void main(String args[]) {
Test ob1 = new Test(2);
Test ob2;
ob2 = ob1.incrByTen();
System.out.println("ob1.a: " + ob1.a);
System.out.println("ob2.a: " + ob2.a);
ob2 = ob2.incrByTen();
System.out.println("ob2.a after second increase: “ + ob2.a);
}
}
```

The output generated by this program is shown here:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

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*

```
// A simple example of recursion.
class Factorial {
// this is a recursive function
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));
}
}
```

- Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional function calls

- Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted

- If this occurs, the Java run-time system will cause an exception

- The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives

```
// Another example that uses recursion.
class RecTest {
int values[];
RecTest(int i) {
values = new int[i];
}
// display array -- recursively
void printArray(int i) {
if(i==0) return;
else printArray(i-1);
System.out.println("[" + (i-1) + "] " + values[i-1]);
}
}
class Recursion2 {
public static void main(String args[]) {
RecTest ob = new RecTest(10);
int i;
for(i=0; i<10; i++) ob.values[i] = i;
ob.printArray(10);
}
}
```

This program generates the following output:

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9
```

Introducing Access Control

- Java's access specifiers are **public**, **private**, and **protected**
- **protected** applies only when inheritance is involved
- When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code
- When a member of a class is specified as **private**, then that member can only be accessed by other members of its class

```
/* This program demonstrates the difference between
   public and private. */
class Test {
int a;              // default access
public int b;    // public access
private int c;   // private access
// methods to access c
void setc(int i) { // set c's value
c = i;
}
int getc() { // get c's value
return c;
}
}
```

```
class AccessTest {
public static void main(String args[]) {
Test ob = new Test();
// These are OK, a and b may be accessed directly
ob.a = 10;
ob.b = 20;
// This is not OK and will cause an error
// ob.c = 100; // Error!
// You must access c through its methods
ob.setc(100); // OK
System.out.println("a, b, and c: " + ob.a + " " +
ob.b + " " + ob.getc());
}
}
```

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
- The most common example of a **static** member is **main( )**
- **main( )** is declared as **static** because it must be called before any objects exist
- Instance variables declared as **static** are, essentially, global variables

Methods declared as **static** have several restrictions:

- They can only call other **static** methods
- They must only access **static** data
- They cannot refer to **this** or **super** in any way

- We can declare a **static** block which gets executed exactly once, when the class is first loaded

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
static int a = 3;
static int b;
static void meth(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[]) {
meth(42);
}
}
```

- As soon as the **UseStatic** class is loaded, all of the **static** statements are run

- First, **a** is set to **3**, then the **static** block executes (printing a message), and finally, **b** is initialized to **a * 4** or **12**

- Then **main( )** is called, which calls **meth( )**, passing **42** to **x**

- If you wish to call a **static** method from outside its class, you can do so using the following general form:

***classname.method*( )**

- Here, *classname* is the name of the class in which the **static** method is declared

```
class StaticDemo {
static int a = 42;
static int b = 99;
static void callme() {
System.out.println("a = " + a);
}
}
class StaticByName {
public static void main(String args[]) {
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
}
}
```

Here is the output of this program:

```
a = 42
b = 99
```

Introducing final

- A variable can be declared as **final**
- Doing so prevents its contents from being modified
- We must initialize a **final** variable when it is declared
- final int FILE_NEW = 1;
- final int FILE_OPEN = 2;

- Variables declared as **final** do not occupy memory on a per-instance basis

- The keyword **final** can also be applied to methods, but its meaning is substantially different than when it is applied to variables

Arrays Revisited

Implemented as objects

The size of an array—that is, the number of elements that an array can hold—is found in its **length** instance variable

```
// This program demonstrates the length array member.
class Length {
public static void main(String args[]) {
int a1[] = new int[10];
int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
int a3[] = {4, 3, 2, 1};
System.out.println("length of a1 is " + a1.length);
System.out.println("length of a2 is " + a2.length);
System.out.println("length of a3 is " + a3.length);
} }
```

Introducing Nested and Inner Classes

- It is possible to define a class within another class
- The scope of a nested class is bounded by the scope of its enclosing class
- If class B is defined within class A, then B is known to A, but not outside of A
- A nested class has access to the members, including private members, of the class in which it is nested

- However, the enclosing class does not have access to the members of the nested class

- There are two types of nested classes: ***static*** and ***non-static***

- A static nested class is one which has the **static** modifier applied
- static innerclass must access its enclosing class by creating an object.

- The most important type of nested class is the *inner* class
- An inner class is a non-static nested class
- It has access to all of the variables and methods of its outer class

```
// Demonstrate an inner class.
class Outer {
int outer_x = 100;
void test() {
Inner inner = new Inner();
inner.display();
}
// this is an inner class
class Inner {
void display() {
System.out.println("display: outer_x = " + outer_x);
}
}
}
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();
}
}
```

- It is important to realize that class **Inner** is known only within the scope of class **Outer**

- The Java compiler generates an error message if any code outside of class **Outer** attempts to instantiate class **Inner**

```
// This program will not compile.
class Outer {
int outer_x = 100;
void test() {
Inner inner = new Inner();
inner.display();
}
// this is an inner class
class Inner {
int y = 10; // y is local to Inner
void display() {
System.out.println("display: outer_x = " + outer_x);
}
}
void showy() {
System.out.println(y); // error, y not known here!
}
}
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();
}
}
```

```
// Define an inner class within a for loop.
class Outer {
  int outer_x = 100;
  void test() {
    for(int i=0; i<10; i++) {
      class Inner {
        void display() {
          System.out.println("display: outer_x = " + outer_x);
        }
      }
```

```
        Inner inner = new Inner();
        inner.display();
      }
    }
}
class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

- While nested classes are not used in most day-to-day programming, they are particularly helpful when handling events in an applet

Using Command-Line Arguments

```
// Display all command-line arguments.
class CommandLine {
public static void main(String args[]) {
for(int i=0; i<args.length; i++)
System.out.println("args[" + i + "]: " +
args[i]);
}
}
```

Execution:

java CommandLine this is a test 100 -1

When you do, you will see the following output:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

Exploring String class

- Creating a String object
- String s="abc".
- class StringDemo {
- public static void main(String args[]) {
- String strOb1 = "First String";
- String strOb2 = "Second String";
- String strOb3 = strOb1 + " and " + strOb2;
- System.out.println(strOb1);
- System.out.println(strOb2);
- System.out.println(strOb3);
- }
- }

```
// Demonstrating some String methods.
class StringDemo2 {
  public static void main(String args[]) {
    String strOb1 = "First String";
    String strOb2 = "Second String";
    String strOb3 = strOb1;
    System.out.println("Length of strOb1: " +
                       strOb1.length());
    System.out.println("Char at index 3 in strOb1: " +
                       strOb1.charAt(3));
```

```
    if(strOb1.equals(strOb2))
      System.out.println("strOb1  == strOb2");
    else
      System.out.println("strOb1  != strOb2");
    if(strOb1.equals(strOb3))
      System.out.println("strOb1  == strOb3");
    else
      System.out.println("strOb1  != strOb3");
  }
}
```

```
// Demonstrate String arrays.
class StringDemo3 {
  public static void main(String args[]) {
    String str[] = { "one", "two", "three" };

    for(int i=0; i<str.length; i++)
      System.out.println("str[" + i + "]: " +
                         str[i]);
  }
```

String constructors

- String(); //empty string creation
- String(char ar[]);
- String(char ar[],int si,int nc);
- String(String ob); //copy one string to other

String member methods

- length();
- char charAt(int where);
- boolean equals(Object str);
- int compareTo(String str)

  Return value <0 if invoking String<str

  Return value>0 if invoking String>str

  =0 if equal
- int indexOf(int ch)
- int indexOf (String sub);
- String toLowerCase()
- String toUppercase();

Sorting array of Strings

```
class SortString {
  static String arr[] = {
    "Now", "is", "the", "time", "for", "all", "good", "men",
    "to", "come", "to", "the", "aid", "of", "their", "country"
  };
  public static void main(String args[]) {
    for(int j = 0; j < arr.length; j++) {
      for(int i = j + 1; i < arr.length; i++) {
        if(arr[i].compareTo(arr[j]) < 0) {
          String t = arr[j];
          arr[j] = arr[i];
          arr[i] = t;
        }
      }
      System.out.println(arr[j]); }}}
```

```
class Check {


public static void main(String[] args) {
String s="this is test string ";
String s1=“”; char c; int i=0,k;
while(true){
k=s.indexOf(' ',i);
if(k==-1) break;
c=s.charAt(i);
c=(char)(c-32);
s1+=c+s.substring(i+1,k)+" ";
i=k+1;
System.out.println(s1);

}
}
}
```

```
class StringReplace {
  public static void main(String args[]) {
    String org = "This is a test. This is, too.";
    String search = "is";
    String sub = "was";
    String result = "";
    int i;
    do { // replace all matching substrings
      System.out.println(org);
      i = org.indexOf(search);
      if(i != -1) {
        result = org.substring(0, i);
        result = result + sub;
        result = result + org.substring(i + search.length());
        org = result;
      }
    } while(i != -1);
  }
}
```

Varargs-Variable length Arguments

- Introduced after jdk5
- Prior to this it was handled in 2 ways
- Method overloading(when maximum no of arguments is known)
- Arrays

```
Class PassArray{
static void vaTest(int v[])
{ System.out.print("No of args  "+v.length);
 for(int x:v) System.out.print(x+" ");
System.out.println();
}
public static void main(String args[]){
int n1[]={10};
int n2[]={1,2,3};
int n3[]={};
vaTest(n1); vaTest(n2) vaTest(n3);}}
```

Using vararg

```
class VarArgs{
static void vaTest(int …v)
{System.out.print("No of args  "+v.length);
 for(int x:v) System.out.print(x+" ");
System.out.println();
}
public static void main(String s[])
{vaTest(10); vaTest(1,2,3); vaTest();
}
}
```

```
//A method can have normal parameters in addition to variable arguments (but last)
class VarArgs2{
static void vaTest(String msg,int …v)
{System.out.print(msg+v.length);
 for(int x:v) System.out.print(x+" ");
System.out.println();
}
public static void main(String s[])
{vaTest("one arg",10); vaTest("three args"1,2,3); vaTest("noargs");
}
}
```

Overloading Varargs

- class VarArgs3{
- static void vaTest(int …v)
- {System.out.print(“vaTest(int …)No of args ”+v.length);
- for(int x:v) System.out.print(x+” “);
- System.out.println();
- }
- static void vaTest(boolean …v)
- {System.out.print(“vaTest(boolean …)No of args ”+v.length);
- for(int x:v) System.out.print(x+” “);

Overloading Varargs

```
System.out.println();
}
static void vaTest(String msg,int …v)
{System.out.print("vaTest(String,int …)No of args  "+msg+v.length);
 for(int x:v) System.out.print(x+" ");
System.out.println();
}
public static void main(String s[])
{vaTest(1,2,3); vaTest("Test",1,2); vaTest(true,false,false);
}
}
```

Ambiguity

- class VarArgs3{
- static void vaTest(int …v)
- {System.out.print(“vaTest(int …)No of args ”+v.length);
- for(int x:v) System.out.print(x+” “);
- System.out.println();
- }
- static void vaTest(boolean …v)
- {System.out.print(“vaTest(boolean …)No of args ”+v.length);
- for(int x:v) System.out.print(x+” “);

```
System.out.println();
}
static void vaTest(String msg,int …v)
{System.out.print("vaTest(String,int …)No of args  "+msg+v.length);
 for(int x:v) System.out.print(x+" ");
System.out.println();
}
public static void main(String s[])
{vaTest(1,2,3); vaTest("Test",1,2); vaTest(true,false,false);
vaTest()// ERROR
}
}
```