PROGRAMMING IN JAVA

RAMAIAH
Institute of Technology

Dr. Manish Kumar
Assistant Professor
Department of Master of Computer Applications
M S Ramaiah Institute of Technology
Bangalore-54

Chapter 7 – A Closer Look at Methods and Classes

Overloading Methods

Here is a simple example that illustrates method overloading:

```
// 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
```

RAMAIAH Institute of Technology

// 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)
}
}
```

This program generates the following output:

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

Overloading Constructors

```
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;
}
}
```

The **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();
```

/* 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
```

RAMAIAH
Institute of Technology

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 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
```

// 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);
}
}
```

RAMAIAH
Institute of Technology

A Closer Look at Argument Passing

```
// 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
```

Returning Objects

```
// 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

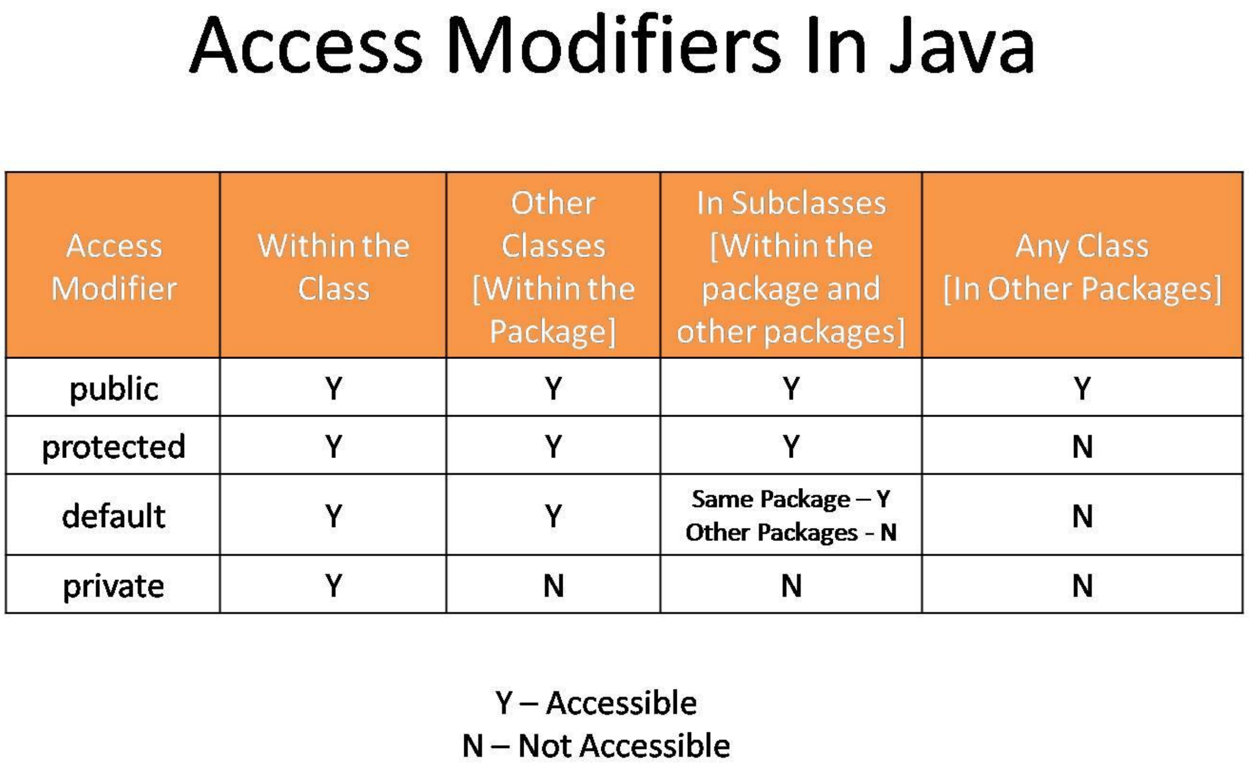
Java supports *recursion*. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be *recursive*.

```
// A simple example of recursion.
class Factorial {
// this is a recursive 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
```

Introducing Access Control



Access Modifiers In Java

Access Modifier	Within the Class	Other Classes [Within the Package]	In Subclasses [Within the package and other packages]	Any Class [In Other Packages]
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	Same Package – Y Other Packages - N	N
private	Y	N	N	N

Y – Accessible
N – Not Accessible

```
/* 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());
}
}
```

```
// This class defines an integer stack that can
hold 10 values.
class Stack {
/* Now, both stck and tos are private. This
means
that they cannot be accidentally or maliciously
altered in a way that would be harmful to the
stack.
*/
private int stck[] = new int[10];
private int tos;
// Initialize top-of-stack
Stack() {
tos = -1;
}
// Push an item onto the stack
void push(int item) {
if(tos==9)
System.out.println("Stack is full.");
else
stck[++tos] = item;
}
// Pop an item from the stack
int pop() {
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
return stck[tos--];
}
}
```

```
class TestStack {
public static void main(String args[]) {
Stack mystack1 = new Stack();
Stack mystack2 = new Stack();
// push some numbers onto the stack
for(int i=0; i<10; i++) mystack1.push(i);
for(int i=10; i<20; i++) mystack2.push(i);
// pop those numbers off the stack
System.out.println("Stack in mystack1:");
for(int i=0; i<10; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<10; i++)
System.out.println(mystack2.pop());
// these statements are not legal
// mystack1.tos = -2;
// mystack2.stck[3] = 100;
}
}
```

Understanding static

- There will be times when you will want to define a class member that will be used independently of any object of that class.
- Normally, a class member must be accessed only in conjunction with an object of its class.
- However, it is possible to create a member that can be used by itself, without reference to a specific instance.
- To create such a member, precede its declaration with the keyword **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 before any objects exist.
- Instance variables declared as **static** are, essentially, global variables.
- When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

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

- They can only directly call other **static** methods.
- They can only directly access **static** data.
- They cannot refer to **this** or **super** in any way. (The keyword **super** relates to inheritance and is described in the next chapter.)

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

Here is the output of the program:

```
Static block initialized.
x = 42
a = 3
b = 12
```

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

classname.method( )

Here is an example. Inside **main( )**, the **static** method **callme( )** and the **static** variable **b** are accessed through their class name **StaticDemo**.

```
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 field can be declared as **final**. Doing so prevents its contents from being modified, making it, essentially, a constant. This means that you must initialize a **final** field when it is declared. You can do this in one of two ways: First, you can give it a value when it is declared. Second, you can assign it a value within a constructor. The first approach is the most common. Here is an example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Introducing Nested and Inner Classes

- It is possible to define a class within another class; such classes are known as *nested classes*.
- The scope of a nested class is bounded by the scope of its enclosing class.
- Thus, if class B is defined within class A, then B does not exist independently 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.
- A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.

- There are two types of nested classes: *static* and *non-static*.
- A static nested class is one that has the **static** modifier applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

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

Output from this application is shown here:

```
display: outer_x = 100
```

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

Exploring the String Class

The first thing to understand about strings is that every string you create is actually an object of type **String**. Even string constants are actually **String** objects. For example, in the statement

```
System.out.println("This is a String, too");
```

Strings can be constructed in a variety of ways. The easiest is to use a statement like this:

```
String myString = "this is a test";
```

Once you have created a **String** object, you can use it anywhere that a string is allowed. For example, this statement displays **myString**:

```
System.out.println(myString);
```

Java defines one operator for **String** objects: **+**. It is used to concatenate two strings. For example, this statement

```
String myString = "I" + " like " + "Java.";
```

results in **myString** containing "I like Java."

The following program demonstrates the preceding concepts:

```
// Demonstrating Strings.
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);
}
}
```

The output produced by this program is shown here:

```
First String
Second String
First String and Second String
```

The **String** class contains several methods that you can use. Here are a few. You can test two strings for equality by using **equals( )**. You can obtain the length of a string by calling the **length()** method. You can obtain the character at a specified index within a string by calling **charAt( )**. The general forms of these three methods are shown here:

boolean equals(*secondStr*)
int length( )
char charAt(*index*)

Here is a program that demonstrates these methods:

RAMAIAH Institute of Technology

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

This program generates the following output:

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
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]);
}
}
```

Here is the output from this program:

```
str[0]: one
str[1]: two
str[2]: three
```

RAMAIAH
Institute of Technology

Using Command-Line Arguments

To access the command-line arguments inside a Java program is quite easy—they are stored as strings in a **String** array passed to the **args** parameter of **main( )**. The first command-line argument is stored at **args[0]**, the second at **args[1]**, and so on. For example, the following program displays all of the command-line arguments that it is called with:

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

Try executing this program, as shown here:

```
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
```

Varargs: Variable-Length Arguments

Beginning with JDK 5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called *varargs* and it is short for *variable-length arguments*. A method that takes a variable number of arguments is called a *variable-arity method*, or simply a *varargs method*.

```
// Use an array to pass a variable number
of
// arguments to a method. This is the
old-style
// approach to variable-length arguments.
class PassArray {
static void vaTest(int v[]) {
System.out.print("Number of args: " +
v.length + " Contents: ");
for(int x : v)
System.out.print(x + " ");
System.out.println();
}
public static void main(String args[])
{
// Notice how an array must be created
to
// hold the arguments.
int n1[] = { 10 };
int n2[] = { 1, 2, 3 };
int n3[] = { };
vaTest(n1); // 1 arg
vaTest(n2); // 3 args
vaTest(n3); // no args
}
}
```

The output from the program is shown here:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

In the program, the method **vaTest( )** is passed its arguments through the array **v**. This old-style approach to variable-length arguments does enable **vaTest( )** to take an arbitrary number of arguments. However, it requires that these arguments be manually packaged into an array prior to calling **vaTest( )**. Not only is it tedious to construct an array each time **vaTest( )** is called, it is potentially error-prone. The varargs feature offers a simpler, better option.

A variable-length argument is specified by three periods (**…**). For example, here is how **vaTest()** is written using a vararg:

```
static void vaTest(int ... v) {
```

```
// Demonstrate variable-length arguments.
class VarArgs {
// vaTest() now uses a vararg.
static void vaTest(int ... v) {
System.out.print("Number of args: " + v.length + " Contents: ");
for(int x : v)
System.out.print(x + " ");
System.out.println();
}
public static void main(String args[])
{
// Notice how vaTest() can be called with a
// variable number of arguments.
vaTest(10); // 1 arg
vaTest(1, 2, 3); // 3 args
vaTest(); // no args
}
}
```

The output from the program is the same as the original version.

RAMAIAH
Institute of Technology

Questions !

RAMAIAH
Institute of Technology

Thank You!