Unit 1

A Closer Look at Methods and Classes

Topics Covered

- Overloading Methods
- Overloading Constructors
- Using Objects as Parameters
- A closer look at Argument Passing
- Returning Objects
- Introducing Access Control
- Understanding Static
- Introducing final
- Arrays
- Exploring the String Class
- Using Command-Line Arguments
- Varargs: Variable Length Arguments

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.

- While overloaded methods may have different return types, ***the return type alone is insufficient*** to distinguish two versions of a method.
- 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.

```
// 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

In the program, test( ) is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters and the fourth takes one double parameter.

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.

```
// 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

This version of OverloadDemo does not define test(int). Therefore, when test( ) is called with an integer argument inside Overload, no matching method is found. However, Java can ***automatically convert an integer into a double*** and this conversion can be used to resolve the call. Therefore, after test(int) is not found, Java elevates i to double and then calls test(double). Java will employ its ***automatic type conversions only if no exact match is found.***

Method overloading supports ***polymorphism*** because it is one way that Java implements the ***"one interface, multiple methods"*** paradigm.

In languages that do not support method overloading, each method must be given a unique name. Consider the absolute value function. In languages that do not support overloading, there are usually three or more versions of this function, each with a slightly different name. For instance, in C, the function ***abs( )*** returns the absolute value of an integer, ***labs( )*** returns the absolute value of a long integer, and ***fabs( )*** returns the absolute value of a floating-point value. Since C does not support overloading, each function has its own name, even though all three functions do essentially the same thing. This makes the situation more complex. Although the underlying concept of each function is the same, we still have three names to remember. This situation does not occur in Java, because each absolute value method can use the same name. Java determines which version of abs( ) to call based upon the type of argument.

The value of overloading is that it allows related methods to be accessed by use of a common name. Through the application of polymorphism, several names can be reduced to one.

But we cannot use the same name to overload unrelated methods. For example, we could use the name **sqr** to create methods that return the square of an integer and the square root of a floating-point value. But these two operations are fundamentally different. Applying method overloading in this manner defeats its original purpose.

Overloading Constructors

```
/* 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

Here the proper overloaded constructor is called based upon the parameters specified when new is executed.

Using Objects as Parameters

It is both correct and common to pass objects to methods. For example, consider the following short program:

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

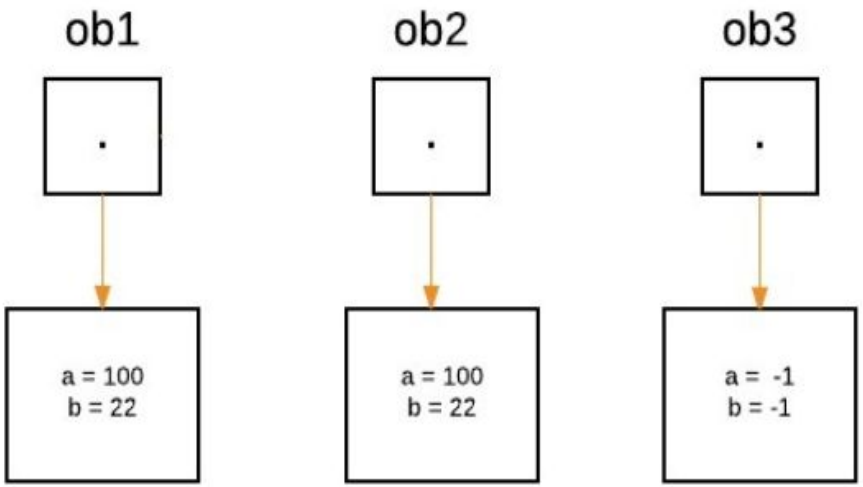
This program generates the following output:

ob1 == ob2: true

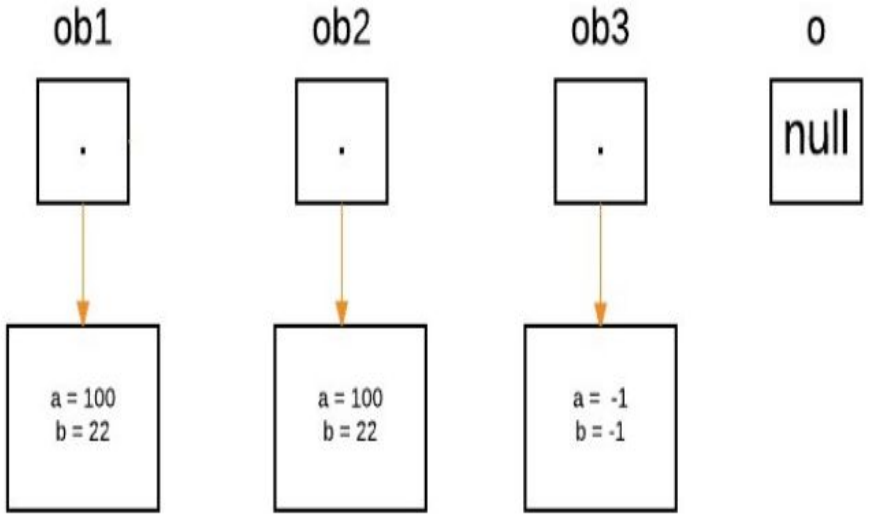
ob1 == ob3: false

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.

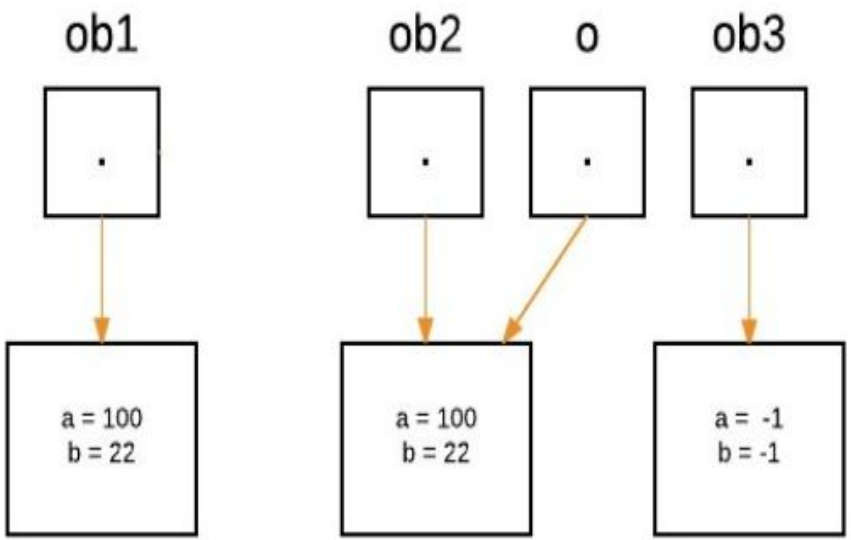
```
ObjectPassDemo ob1 = new ObjectPassDemo(100, 22);
ObjectPassDemo ob2 = new ObjectPassDemo(100, 22);
ObjectPassDemo ob3 = new ObjectPassDemo(-1, -1);
```



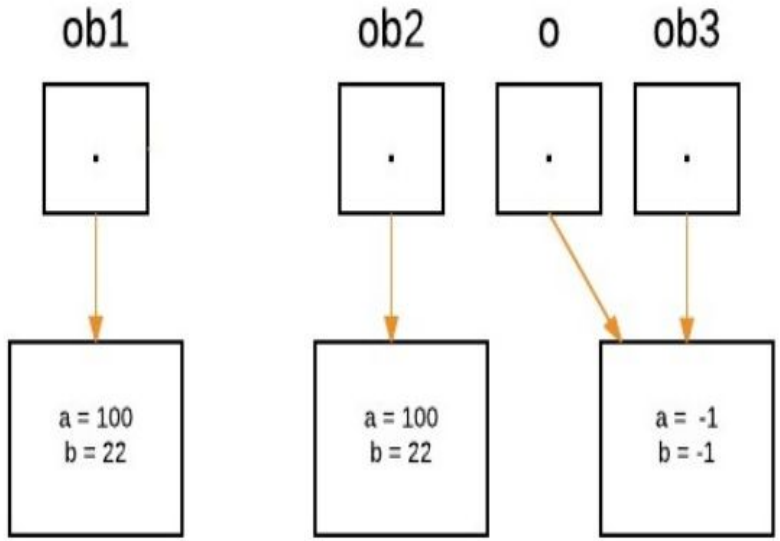

```
boolean equalTo(ObjectPassDemo o);
```



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




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



One of the most common uses of object parameters involves constructors. Frequently, we will want to construct a new object so that it is initially the same as some existing object. To do this, we must define a constructor that takes an object of its class as a parameter. For example, the following version of Box allows one object to initialize another:

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

The output of this program is

Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of cube is 343.0
Volume of clone is 3000.0

A Closer Look at Argument Passing

In general, there are two ways we can pass an argument to a subroutine.

- The first way is **call-by-value**. This approach copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.
- The second way an argument can be passed is **call-by-reference**. In this approach, 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.

Although Java uses **call-by-value** to pass all arguments, the precise effect differs between whether a primitive type or a reference type is passed.

When we pass a primitive type to a method, it is passed by value. Thus, a copy of the argument is made and what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

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

The operations that occur inside meth( ) have no effect on the values of a and b used in the call; their values here did not change to 30 and 10.

When we pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively ***call-by-reference***. When we create a variable of a class type, we are only creating a reference to an object. Thus, when we pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects act as if they are passed to methods by use of call-by-reference. Changes to the object inside the method do affect the object used as an argument. For example, consider the following program:

```
// 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

The actions inside meth( ) have affected the object used as an argument.

Introducing Access Control

- Encapsulation links data with the code that manipulates it.
- However, encapsulation provides another important attribute: **access control**.
- Through encapsulation, we can control what parts of a program can access the members of a class.
- By controlling access, we can prevent misuse.
- For example, allowing access to data only through a well-defined set of methods, we can prevent the misuse of that data.
- Thus, when correctly implemented, a class creates a “black box” which may be used, but the inner workings of which are not open to tampering.

- How a member can be accessed is determined by the access modifier attached to its declaration.
- Java supplies a rich set of access modifiers.
- Java's access modifiers are **public, private**, and **protected**.
- Java also defines a default access level.
- **protected** applies only when inheritance is involved.
- When a member of a class is modified by **public**, 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.
- So **main( )** is always preceded by the public modifier. It is called by code that is outside the program—that is, by the Java run-time system.
- *When no access modifier is used, then by default the member of a class is **public** within its own package, but cannot be accessed outside of its package.*

- Usually, we will want to restrict access to the data members of a class—allowing access only through methods.
- Also, there will be times when we will want to define methods that are private to a class.

- An access modifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here is an example:

```
public int i;
private double j;
private int myMethod(int a, char b) { //…
```

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

In this program, inside the Test class, **a** uses default access, which is same as specifying public. **b** is explicitly specified as public. Member **c** is given private access. This means that it cannot be accessed by code outside of its class. So, inside the **AccessTest** class, **c** cannot be used directly. It must be accessed through its public methods: **setc( )** and **getc( )**.

ob.c = 100; // Error!

We would not be able to compile this program because of the access violation.

To see how access control can be applied to a more practical example, consider the following improved version of the Stack class

```
// 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--];
  }
}
```

Now both stck, which holds the stack and tos, which is the index of the top of the stack, are specified as private. This means that they cannot be accessed or altered except through push( ) and pop( ). Making tos private, for example, prevents other parts of program from inadvertently setting it to a value that is beyond the end of the stck array.

Understanding static

- There will be times when we 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.
- We 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.

If we need to do computation in order to initialize static variables, we can declare a static block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a static method, some static variables and a static initialization block

```
// 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, which prints a message and then initializes **b** to a*4 or 12. Then main( ) is called, which calls meth( ), passing 42 to **x**. The three println( ) statements refer to the two static variables **a** and **b**, as well as to the local variable **x**.

Here is the output of the program:

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

To call a static method from outside its class, we 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.

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.
- We can do this in one of two ways:
  - ❖ First, we can give it a value when it is declared.
  - ❖ Second, we 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;
```

- Subsequent parts of the program can now use ***FILE_OPEN***, etc., as if they were constants, without fear that a value has been changed.
- It is a common coding convention to choose all uppercase identifiers for **final** fields
- In addition to fields, both method parameters and local variables can be declared **final**.
- Declaring a parameter **final** prevents it from being changed within the method.
- Declaring a local variable **final** prevents it from being assigned a value more than once.
- The keyword **final** can also be applied to methods and classes
- **final** methods prevents the concept of method overriding
- **final** class prevents inheritance

```
class Bike
{
  final int speedlimit = 90;//final variable
  void run()
{
   speedlimit=400;
 }
  public static void main(String args[])
{
  Bike obj=new  Bike();
  obj.run();
  }
 }
```

The output of the program gives following error

Error: cannot assign a value to final variable speedlimit, speedlimit=400;

Arrays

- Arrays are 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.
- All arrays have this variable and it will always hold the size of the array.

The program demonstrates this property:

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

This program displays the following output:

length of a1 is 10
length of a2 is 8
length of a3 is 4

Here the size of each array is displayed. The value of ***length*** has nothing to do with the number of elements that are actually in use. It only reflects the number of elements that the array is designed to hold.

Exploring the String Class

- String is the most commonly used class in Java's class library.
- The first thing to understand about strings is that every string we 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");

  the string ***"This is a String, too"*** is a String object.

- The second thing to understand about strings is that objects of type String are immutable; once a String object is created, its contents cannot be altered. While this may seem like a serious restriction, it is not, for two reasons:
  - ★ If we need to change a string, we can always create a new one that contains the modifications.
  - ★ Java defines peer classes of String, called ***StringBuffer*** and ***StringBuilder***, which allow strings to be altered, so all of the normal string manipulations are still available in Java.

Note: The StringBuffer and StringBuilder class provides an alternative to String Class, as it creates a mutable sequence of characters.

- 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 we have created a ***String*** object, we 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 we can use.

- We can test two strings for equality by using equals( ).
- We can obtain the length of a string by calling the length( ) method.
- We 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:

```
// 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

We can have arrays of strings, just like we can have arrays of any other type of object. For example:

```
// 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

Using Command-Line Arguments

- Sometimes we will want to pass information into a program when we run it.
- This is accomplished by passing command-line arguments to main( ).
- A command-line argument is the information that directly follows the program's name on the command line when it is executed.
- 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.
- A Java application can accept any number of arguments from the command line. This allows the user to specify configuration information when the application is launched.

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
```

Note: All command-line arguments are passed as strings.

Varargs: Variable-Length Arguments

- Java has 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.
- Situations that require that a variable number of arguments be passed to a method are not unusual. For example, a method that opens an Internet connection might take a username, password, filename, protocol and so on, but supply defaults if some of this information is not provided. In this situation, it would be convenient to pass only the arguments to which the defaults did not apply.

- In varargs feature,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) {

This syntax tells the compiler that ***vaTest( )*** can be called with zero or more arguments. As a result, v is implicitly declared as an array of type int[ ]. Thus, inside ***vaTest( )***, v is accessed using the normal array syntax.

```
// 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 shown here:

Number of args: 1 Contents: 10

Number of args: 3 Contents: 1 2 3

Number of args: 0 Contents:

There are two important things to notice about this program.

- First, inside ***vaTest( )***, ***v*** is operated on as an array. The ...syntax simply tells the compiler that a variable number of arguments will be used and that these arguments will be stored in the array referred to by ***v***.

- Second, in ***main( )***, ***vaTest( )*** is called with different numbers of arguments, including no arguments at all. The arguments are automatically put in an array and passed to ***v***. In the case of no arguments, the length of the array is zero.

- A method can have “normal” parameters along with a variable-length parameter. However, the variable-length parameter must be the last parameter declared by the method. For example, this method declaration is perfectly acceptable:

  int doIt(int a, int b, double c, int ... vals) {

  In this case, the first three arguments used in a call to doIt( ) are matched to the first three parameters. Then, any remaining arguments are assumed to belong to vals.

- Remember, the varargs parameter must be last. For example, the following declaration is incorrect:

  int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Error!

  Here, there is an attempt to declare a regular parameter after the varargs parameter, which is illegal.

- There is one more restriction to be aware of: there must be only one varargs parameter. For example, this declaration is also invalid:

  int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Error!

  The attempt to declare the second varargs parameter is illegal.

Overloading Vararg Methods

We can overload a method that takes a variable-length argument. For example, the following program overloads vaTest( ) three times:

```
// Varargs and overloading.
class VarArgs3 {

  static void vaTest(int ... v) {
    System.out.print("vaTest(int ...): " +
                     "Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

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

```
  static void vaTest(boolean ... v) {
    System.out.print("vaTest(boolean ...) " +
                     "Number of args: " + v.length +
                     " Contents: ");

    for(boolean x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  static void vaTest(String msg, int ... v) {
    System.out.print("vaTest(String, int ...): " +
                     msg + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String args[])
  {
    vaTest(1, 2, 3);
    vaTest("Testing: ", 10, 20);
    vaTest(true, false, false);
  }
}
```

The output produced by this program is shown here:

vaTest(int ...): Number of args: 3 Contents: 1 2 3
vaTest(String, int ...): Testing: 2 Contents: 10 20
vaTest(boolean ...) Number of args: 3 Contents: true false false

This program illustrates both ways that a varargs method can be overloaded.

- First, the types of its vararg parameter can differ. This is the case for vaTest(int ...) and vaTest(boolean ...). Remember, the ... causes the parameter to be treated as an array of the specified type.In this case, Java uses the type difference to determine which overloaded method to call.
- The second way to overload a varargs method is to add one or more normal parameters. This is what was done with vaTest(String, int ...). In this case, Java uses both the number of arguments and the type of the arguments to determine which method to call.

Varargs and Ambiguity

Somewhat unexpected errors can result when overloading a method that takes a variable-length argument. These errors involve ambiguity because it is possible to create an ambiguous call to an overloaded varargs method.
For example, consider the following program:

```
// Varargs, overloading, and ambiguity.
//
// This program contains an error and will
// not compile!
class VarArgs4 {

  static void vaTest(int ... v) {
    System.out.print("vaTest(int ...): " +
                     "Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

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

```
static void vaTest(boolean ... v) {
  System.out.print("vaTest(boolean ...) " +
                   "Number of args: " + v.length +
                   " Contents: ");

  for(boolean x : v)
    System.out.print(x + " ");

  System.out.println();
}

public static void main(String args[])
{

     vaTest(1, 2, 3); // OK
     vaTest(true, false, false); // OK

     vaTest(); // Error: Ambiguous!
   }
 }
```

In this program, the overloading of ***vaTest( )*** is perfectly correct. However, this program will not compile because of the following call:

vaTest(); // Error: Ambiguous!

Because the vararg parameter can be empty, this call could be translated into a call to ***vaTest(int ...)*** or ***vaTest(boolean ...)***. Both are equally valid. Thus, the call is inherently ambiguous.

Here is another example of ambiguity. The following overloaded versions of ***vaTest( )*** are inherently ambiguous even though one takes a normal parameter:

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

static void vaTest(int n, int ... v) { // …

Although the parameter lists of ***vaTest( )*** differ, there is no way for the compiler to resolve the following call:

vaTest(1)

Does this translate into a call to ***vaTest(int ...)***, with one varargs argument, or into a call to ***vaTest(int, int ...)*** with no varargs arguments? There is no way for the compiler to answer this question. Thus, the situation is ambiguous.

Java Program to take input from user - Program 1

```
import java.util.Scanner; // import the Scanner class

class Main
{
  public static void main(String[] args)
{
    Scanner myObj = new Scanner(System.in);
    String userName;

    // Enter username and press Enter
    System.out.println("Enter username");
    userName = myObj.nextLine();
    System.out.println("Username is: " + userName);
  }
   }
```

Program 2

```
import java.util.Scanner;
class Main
{
  public static void main(String[] args)
 {
    Scanner myObj = new Scanner(System.in);
    System.out.println("Enter name, age and salary:");
    // String input
    String name = myObj.nextLine();
    // Numerical input
    int age = myObj.nextInt();
    double salary = myObj.nextDouble();
    // Output input by user
    System.out.println("Name: " + name);
    System.out.println("Age: " + age);
    System.out.println("Salary: " + salary);
  }}
```