

**Dr. Ganga Holi, Professor and Head, Dept of CSE- ICB**

**Module 1 Notes**

**Object Oriented Programming with JAVA**

**BCS306A -III Sem ESC**

**Module-1**

**Introduction to Java**

**The Bytecode**: The Bytecode is a highly optimized set ofinstructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM). In essence, the original JVM was designed as an interpreter for bytecode.

Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all understand the same Java bytecode.

If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs.

**The Java Buzzwords**

The key considerations were summed up by the Java team in the following list of buzzwords:

* **Simple:** Java was designed to be easy for the professionalprogrammer to learn and use effectively.

If already understand the basic concepts of object-oriented programming, learning Java will be even easier.

* **Secure:** Every time you download a “normal” program, you aretaking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources.

Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer.

* + **Portable:** Portability is a major aspect of the Internet because thereare many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems. For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet.
  + **Object-oriented:** Although influenced by its predecessors, Java wasnot designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects.
* **Robust:** The multi-plat-formed environment of the Web placesextraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java.
* **Multithreaded:** Java was designed to meet the real-worldrequirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multi-process synchronization that enables you to construct smoothly running interactive systems
  + **Architecture-Neutral :** A central issue for the Java designers wasthat of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. Their goal was “write once; run anywhere, anytime, forever.” To a great extent, this goal was accomplished.
* **Interpreted & High performance:** Java enables the creation of

cross-platform programs by compiling into an intermediate

representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance.

The Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

* **Distributed:** Java is designed for the distributed environment of theInternet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.
  + **Dynamic:** Java programs carry with them substantial amounts ofrun-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically pdated on a running system.

**Object-Oriented Programming**

Object-oriented programming (OOP) is at the core of Java. In fact, all Java programs are to at least some extent object-oriented.

**Abstraction**

Abstraction in Java or Object oriented programming is a way to segregate implementation from interface and one of the five fundamentals along with Encapsulation, Inheritance, Polymorphism, Class and Object.

* + An essential component of object-oriented programming is

Abstraction

* Humans manage complexity through abstraction.
  + For example, people do not think a car as a set of tens and thousands of individual parts. They think of it as a well-defined object with its own unique behavior.
  + This abstraction allows people to use a car ignoring all details of how the engine, transmission and braking systems work.
* In computer programs the data from a traditional process-oriented program can be transformed by abstraction into its component objects.
* A sequence of process steps can become a collection of messages between these objects. Thus each object describes its own behaviour.

**The Three OOP Principles**

All object-oriented programming languages provide mechanisms that help to implement the object-oriented model. They are encapsulation, inheritance, and polymorphism.

**Encapsulation**

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.

* One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper.
* Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. • To relate this to the real world, consider the automatic transmission on an automobile.

* It encapsulates hundreds of bits of information about your engine, such as how much we are accelerating, the pitch of the surface we are on, and the position of the shift.
  + The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects.

**Polymorphism**

Polymorphism (from the 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). We 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 Java we 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. This means that it is possible to design a generic interface to a group of related activities.
  + This helps reduce complexity by allowing the same interface to be used to specify a general class of action.
  + Polymorphism allows us to create clean, sensible, readable, and resilient code.

**A First Simple Program**

class Example

{

// Your program begins with a call to main().

public static void main(String args[])

{

System.out.println("This is a simple Java program.");

}

}

**Entering the program**

* In Java, a source file is a text file that contains one or more class definitions. The Java compiler requires that a source file use the .java filename extension.
* In Java, all code must reside inside a class. By convention, the name of that class should match the name of the file that holds the program. You should also make sure that the capitalization of the filename matches the class name.
* The reason for this is that Java is case-sensitive. At this point, the convention that filenames correspond to class names may seem arbitrary

**Compiling the Program**

C:\>javac Example.java

The javac compiler creates a file called Example.class that contains the bytecode version of the program.To actually run the program, you must use the Java application launcher, called java. To do so, pass the class name Example as a command-line argument, as shown here: C:\>java Example When the program is run, the following output is displayed: This is a simple Java program. When Java source code is compiled, each individual class is put into its own output file named after the class and using the .class extension.

**A Closer Look at the First Sample Program**

The program begins with the following lines:

/\* This is a simple Java program. Call this file "Example.java". \*/ - >This is a comment.

The contents of a comment are ignored by the compiler. Instead, a comment describes or explains the operation of the program to anyone who is reading its source code.

Java supports three styles of comments. The one shown at the top of the program is called a multiline comment. This type of comment must begin with /\* and end with \*/. Anything between these two comment symbols is ignored by the compiler. As the name suggests, a multiline comment may be several lines long.

The next line of code in the program is shown here: class Example { This line uses the keyword class to declare that a new class is being defined. Example is an identifier that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace (}).

The next line in the program is the single-line comment, shown here:

* Your program begins with a call to main(). This is the second type of comment supported by Java. A single-line comment begins with a
* and ends at the end of the line.

The next line of code is shown here: public static void main(String args[]) { This line begins the main( ) method. As the comment preceding it suggests, this is the line at which the program will begin executing. All Java applications begin execution by calling main( )

The **public** keyword is an access specifier, which allows the programmer to control the visibility of class members. When a class member is preceded by public, then that member may be accessed by code outside the class in which it is declared.

In this case, **main( )** must be declared as public, since it must be called by code outside of its class when the program is started. The keyword **static** allows main( ) to be called without having to instantiate a particular instance of the class. Any information that you need to pass to a method is received by variables specified within the set of parentheses that follow the name of the method. These variables are called parameters. If there are no parameters required for a given method, you still need to include the empty parentheses. In main( ), there is only one parameter, albeit a complicated one.**String args[ ]** declares a parameter named args, which is an array of instances of the class String.

The next line of code is shown here. Notice that it occurs inside main( ). System.out.println("This is a simple Java program."); This line outputs the string “This is a simple Java program.” followed by a new line on the screen. Output is actually accomplished by the built-in println( ) method. In this case, println( ) displays the string which is passed to it.

**A Second Short Program**

class Example2

{

public static void main(String args[])

{

int num; // this declares a variable called num

num = 100; // this assigns num the value 100

System.out.println("This is num: " + num);

num = num \* 2;

System.out.print("The value of num \* 2 is ");

System.out.println(num);

}

}

When you run this program, you will see the following output:

**Output is :**

This is num: 100

The value of num \* 2 is 200

**Chapter : Data Types, Variables, and Arrays**

The Primitive Types Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean.

The primitive types are also commonly referred to as simple types, and both terms will be used in this book. These can be put in four groups:

* Integers This group includes byte, short, int, and long, which are for whole-valued signed numbers. • Floating-point numbers This group includes float and double, which represent numbers with fractional precision.
  + Characters This group includes char, which represents symbols in a character set, like letters and numbers.
  + Boolean This group includes boolean, which is a special type for representing true/false values.

Java defines four integer types: byte, short, int, and long. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers. Many other computer languages support both signed and unsigned integers.



The width and ranges of these integer types vary widely, as shown in this table:

|  |  |  |
| --- | --- | --- |
| Name | Width in bits | Range |
| Long | 64 | 9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| Int | 32 | -2,147,483,648 to 2,147,483,647 |
| Short | 16 | -32,768 to 32,767 |
| Byte | 8 | -256 to 255 |

**Integers**

The most commonly used integer type is int. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type int are commonly employed to control loops and to index arrays.

**Byte**

The smallest integer type is byte. This is a signed 8-bit type that has a range from –128 to 127. Variables of type byte are especially useful when you’re working with a stream of data from a network or file.

For example, the following declares two byte variables called b and c:

byte b, c;

**short**

short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used Java type. Here are some examples of short variable declarations:

short s; short t;

**long:** Is a signed 64-bit type and is useful for those occasions where an inttype is not large enough to hold the desired value. The range of a long is quite large. This makes it useful when big, whole numbers are needed.

**Floating-Point Types**

Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. For example,

calculations such as square root, or transcendentals such as sine and cosine, result in a value whose precision requires a floating-point type.

There are two kinds of floating-point types, float and double, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

|  |  |  |
| --- | --- | --- |
| Name | Width in Bits | Approximate Range |
|  |  |  |
| Double | 64 | 4.9e–324 to 1.8e+308 |
|  |  |  |
| Float | 32 | 1.4e–045 to 3.4e+038 |

**float**

The type float specifies a single-precision value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type float are useful when you need a fractional component, but don’t require a large degree of precision. For example, float can be useful when representing dollars and cents. Here are some example float variable declarations:

float hightemp, lowtemp;

**double**

Double precision, as denoted by the double keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as sin( ), cos( ), and sqrt( ), return double values.

Here is a short program that uses double variables to compute the area of a circle: // Compute the area of a circle.

class Area {

public static void main(String args[]) {

double pi, r, a;

r = 10.8; // radius of circle pi = 3.1416; // pi, approximately

a = pi \* r \* r; // compute area

System.out.println("Area of circle is " + a);

}

}

**Characters**

In Java, the data type used to store characters is char. However, C/C++ programmers beware: char in Java is not the same as char in C or C++. In C/C++, char is 8 bits wide. This is not the case in Java. Instead, **Java uses Unicode to represent characters. Unicode defines a fully international character set that can represent all of the characters found in all human languages.**

**Java char is a 16-bit type.** The range of a char is 0 to 65,536. There are no negative chars. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255.

class CharDemo

{ public static void main(String args[])

{ char ch1, ch2; ch1 = 88; // code for X

ch2 = 'Y';

System.out.print("ch1 and ch2: ");

System.out.println(ch1 + " " + ch2);

}

}

This program displays the following output: ch1 and ch2: X Y Notice that ch1 is assigned the value 88, which is the ASCII (and Unicode) value that

corresponds to the letter X. As mentioned, the ASCII character set occupies the first 127 values in the Unicode character set.

**Booleans**

Java has a primitive type, called boolean, for logical values. It can have only one of two possible values, true or false. This is the type returned by all relational operators, as in the case of a < b. boolean is also the type required by the conditional expressions that govern the control statements such as if and for. Here is a program that demonstrates the boolean type:

class BoolTest

{ public static void main(String args[])

{ boolean b; b = false;

System.out.println("b is " + b);

b = true;

System.out.println("b is " + b);

// a boolean value can control the if statement

if(b)

System.out.println("This is executed.");

b = false;

if(b)

System.out.println("This is not executed.");

// outcome of a relational operator is a boolean value

System.out.println("10 > 9 is " + (10 > 9));

}

}

The output generated by this program is shown here:

b is false

b is true

This is executed. 10 > 9 is true

There are three interesting things to notice about this program. First, as you can see, when a boolean value is output by println( ), “true” or “false” is displayed. Second, the value of a boolean variable is sufficient, by itself, to control the if statement. There is no need to write an if statement like this: if(b == true) .

**Literals**

**Integer Literals**

Integer literals create an int value, which in Java is a 32-bit integer value. Since Java is strongly typed, you might be wondering how it is possible to assign an integer literal to one of Java’s other integer types, such as byte or long, without causing a type mismatch error. Fortunately, such situations are easily handled. When a literal value is assigned to a byte or short variable, no error is generated if the literal value is within the range of the target type.

**An integer literal can always be assigned to a long variable. However, to specify a long literal, you will need to explicitly tell the compiler that the literal value is of type long. You do this by appending an upper- or lowercase L to the literal. For example, 0x7ffffffffffffffL or 9223372036854775807L is the largest long. An integer can also be assigned to a char as long as it is within range.**

Beginning with JDK 7, you can also specify integer literals using binary. To do so, prefix the value with 0b or 0B. For example, this specifies the decimal value 10 using a binary

literal:

int x = 0b1010;

Among other uses, the addition of binary literals makes it easier to enter values used as bitmasks. In such a case, the decimal (or hexadecimal) representation of the value does not visually convey its meaning relative to its use. The binary literal does.

Also beginning with JDK 7, you can embed one or more underscores in an integer literal. Doing so makes it easier to read large integer literals. When the literal is compiled, the underscores are discarded. For example, given

int x = 123\_456\_789;

the value given to x will be 123,456,789. The underscores will be ignored.

Underscores can only be used to separate digits. They cannot come at the beginning or the end of a literal. It is, however, permissible for more than one underscore to be used between two digits.

For

example, this is valid:

int x = 123\_\_\_456\_\_\_789;

**For example,**

**binary values are often visually grouped in four-digits units, as shown here:**

**int x = 0b1101\_0101\_0001\_1010;**

**Floating-Point Literals**

**Scientific notation uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied. The exponent is indicated** **by an E or e followed by a decimal number, which can be positive or negative. Examples include 6.022E23, 314159E–05, and 2e+100.**

Floating-point literals in Java default to double precision. To specify a float literal, you must append an F or f to the constant. You can also explicitly specify a double literal by appending a D or d.

**Boolean Literals**

Boolean literals are simple. There are only two logical values that a boolean value can have, true and false. The values of true and false do not convert into any numerical representation.

The true literal in Java does not equal 1, nor does the false literal equal 0. In Java, they can only be assigned to variables declared as boolean, or used in expressions with Boolean operators.

**Character Literals**

A literal character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as ‘a’, ‘z’, and ‘@’. For characters that are impossible to enter directly, there are several escape sequences that allow you to enter the character you need, such as ‘\’’ for the single-quote character itself and ‘\n’for the newline character. There is also a mechanism for directly entering the value of a character in octal or hexadecimal. For octal notation, use the backslash followed by the three-digit number. For example, ‘\141’ is the letter ‘a’. For hexadecimal, you enter a backslash-u (\u), then exactly four hexadecimal digits. For example, ‘\u0061’is the ISO-Latin-1 ‘a’ because the top byte is zero. ‘\ua432’is a Japanese Katakana character.

Example:

\ddd Octal character (ddd)

\uxxxx Hexadecimal Unicode character (xxxx)

\' Single quote

\" Double quote

[\\Backslash](file:///\\Backslash)

**String Literals**

String literals in Java are specified like they are in most other languages—by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are

“Hello World”

“two\nlines”

“\”This is in quotes\”“

**One important thing to note about Java strings is that they must begin and end on the same line**. There is no line-continuation escape sequence as there is in some other languages.

**Variables:**

**The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer.** In addition, all variables have a scope, which defines **their visibility, and a lifetime.** In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

type identifier [ = value][, identifier [= value] ...];

The *type* is one of Java‘s atomic types, or the name of a class or interface. The *identifier* is the name of the variable. You can initialize the variable by specifying an equal sign and a value. To declare more than one variable of the specified type, use a comma separated list.

Here are several examples of variable declarations of various types.

Note that some include an initialization.

int a, b, c; // declares three ints, a, b, and c. int d = 3, e, f = 5; // declares three more ints byte z = 22; // initializes z.

double pi = 3.14159; // declares an approximation of pi.

char x = 'x'; // the variable x has the value 'x'.

**Dynamic Initialization**

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

// Demonstrate dynamic initialization.

class DynInit {

public static void main(String args[]) {

double a = 3.0, b = 4.0;

// c is dynamically initialized

double c = Math.sqrt(a \* a + b \* b);

System.out.println("Hypotenuse is " + c);

}

}

Here, three local variables—a, b, and c—are declared. The first two, a and b, are initialized by constants. However, c is initialized dynamically to the length of the hypotenuse (using the Pythagorean theorem). The program uses another of Java’s built-in methods, sqrt(), which is a member of the Math class, to compute the square root of its argument. The key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

**The Scope and Lifetime of Variables:**

All of the variables used till now have been declared at the start of the

**main( )** method. However, Java allows variables to be declared withinany block. A block is begun with an opening curly brace and ended by a closing curly brace. A block defines a *scope.* Thus, each time you start a new block, you are creating Scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

Most other computer languages define two general categories of scopes: **global and local**. The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method‘s scope. variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope.

This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.

To understand the effect of nested scopes, consider the following program:

//demonstrate block scope.

class Scope

{ public static void main(String args[])

{ int x; // known to all code within main x = 10;

if(x == 10)

{ // start new scope

int y = 20; // known only to this block

System.out.println(“x : "+x);

System.out.println(“y : "+y);

x = y \* 2;

}

y = 100; // Error! y not known here

System.out.println("x is " + x); //

}

}

Check the output???

As the comments indicate, the variable **x** is declared at the start of **main( )**‘s scope and is accessible to all subsequent code within **main( )**. Within the **if** block, **y** is declared. Since a block defines a scope, **y** isonly visible to other code within its block. This is why outside of its block, the line **y = 100;** is commented out. If you remove the leading comment symbol, a compile-time error will occur, because **y** is not visible outside of its block. Within the **if** block, **x** can be used because code within a block (that is, a nested scope) has access to variables declared by an enclosing scope.

Here is another important point to remember: **variables are created when their scope is entered and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope.** Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope. If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered.

For example, consider the next program.

* Demonstrate lifetime of a variable. class Lifetime

{ public static void main(String args[])

{ int x;

for(x = 0; x < 3; x++)

{ int y = -1; // y is initialized each time block is entered

System.out.println("y is: " + y);

// this always prints -1

y = 100;

System.out.println("y is now: " + y);

}

}

}

The output generated by this program is shown here:

y is: -1

y is now: 100

y is: -1

y is now: 100

y is: -1

y is now: 100

As you can see, **y** is always reinitialized to –1 each time the inner **for** loop is entered. Even though it is subsequently assigned the value 100, this value is lost. One last point: Although blocks can be nested, cannot declare a variable to have the same name as one in an outer scope.

**Type Conversion and Casting:**

We can assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an int value to a long variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed.

For instance, there is no conversion defined from double to byte. But it is possible for conversion between incompatible types. To do so, you must use a cast, which performs an explicit conversion between incompatible types.

**Java‘s Automatic Conversions**

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are satisfied:

* The two types are compatible.
* The destination type is larger than the source type.

When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required. For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, the numeric types are not compatible with char or boolean. Also, char and boolean are not compatible with each other.

Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, or long.

**Casting Incompatible Types**

The automatic type conversions are helpful, they will not fulfil all needs. For example, if We want to assign an int value to a byte variable. This conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is sometimes called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type.

To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion. It has this general form:

**(target-type) value**

Here, target-type specifies the desired type to convert the specified value to.

Example:

int a;

byte b;

// ...

**b = (byte) a;**

A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation. As integers do not have fractional components so, when a floating-point value is assigned to an integer type, the fractional component is lost.

Program:

class Conversion

{

public static void main(String args[])

{ byte b;

int i = 257;

double d = 323.142;

System.out.println("\nConversion of int to byte.");

b = (byte) i;

System.out.println("i and b " + i + " " + b);

System.out.println("\nConversion of double to int.");

i = (int) d;

System.out.println("d and i " + d + " " + i);

System.out.println("\nConversion of double to byte.");

b = (byte) d;

System.out.println("d and b " + d + " " + b);

}

}

**Output:**

Conversion of int to byte.

i and b 257 1

Conversion of double to int.

d and i 323.142 323

Conversion of double to byte.

d and b 323.142 67

byte b = 50;

b = (byte)(b \* 2);

**Arrays:**

An *array* is a group of similar-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its

index. Arrays offer a convenient means of grouping related information.

One-Dimensional Arrays

A *one-dimensional array* is a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one dimensional array declaration is

*type var-name*[ ];

Here, *type* declares the base type of the array. For example, the following declares an

array named **month** with the type ―array of int‖:

int month [];

Although this declaration establishes the fact that **month** is an array variable, no array actually exists. In fact, the value of **month** is set to **null**, which represents an array with no value. To link **month** with anactual, physical array of integers, you must allocate one using **new** and assign it to **month**. **new** is a special operator that allocates memory. The general form of **new** as it applies to one-dimensional arrays appears as follows:

*array-var* = new *type*[*size*];

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *array-var* is the array variable that is linked to the array. That is, to use **new** to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by **new** will automatically be initialized to zero.

This example allocates a 12-element array of integers and links them to **month**

month = new int[12];

After this statement executes, **month** will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero. Another way to declare an arrayin single step is type arr-name=new type[size];

Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An *array initializer* is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use **new**.

For example, to store the number of days in each month, we do as follows

* An improved version of the previous program. class AutoArray

{ public static void main(String args[])

{ int month[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,30, 31 };

System.out.println("April has " + month[3] + " days.");

}

}

When you run this program, in the output it prints the number of days in April. As mentioned, Java array indexes start with zero, so the number of days in April is **month[3]**  so 30.

Here is one more example that uses a one-dimensional array. It finds the average of a set of numbers.

* Average an array of values. class Average

{ public static void main(String args[])

{ double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};

double result = 0;

int i;

for(i=0; i<5; i++)

result = result + nums[i];

System.out.println("Average is " + result / 5);

}

}

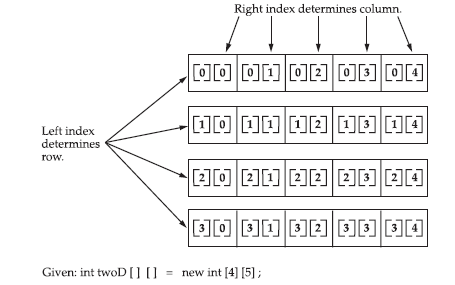
Output: Average is:12.3

**Multidimensional Arrays**

In Java, multidimensional arrays are actually arrays of arrays. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called **twoD**.

int twoD[][] = new int[4][5];

This allocates a 4 by 5 array and assigns it to **twoD**.



program :

* Demonstrate a two-dimensional array. class TwoDArray

{ public static void main(String args[])

{ int twoD[][]= new int[4][5]; int i, j, k = 0;

for(i=0;i<4;i++)

{ for(j=0;j<5;j++)

{ twoD[i][j] = k;

k++;

}

}

for(i=0; i<4; i++)

{ for(j=0; j<5; j++)

{

System.out.print(twoD[i][j] + " ");

}

System.out.println();

}

}

}

Output:

0 1 2 3 4

5 6 7 8 9

10 11 12 13 14

15 16 17 18 19

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. **You can allocate the remaining dimensions separately. We can allocates the second dimension manually.**

int twoD[][] = new int[4][];

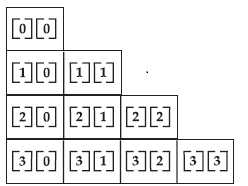
twoD[0] = new int[5];

twoD[1] = new int[5];

twoD[2] = new int[5];

twoD[3] = new int[5];

we can creates a two dimensional array in which the sizes of the second dimension are unequal.



* Manually allocate differing size second dimensions. class TwoDAgain

{ public static void main(String args[])

{ int twoD[][] = new int[4][];

twoD[0] = new int[1];

twoD[1] = new int[2];

twoD[2] = new int[3];

twoD[3] = new int[4];

int i, j, k = 0;

for(i=0; i<4; i++)

{ for(j=0; j<i+1; j++)

{ twoD[i][j] = k; k++;

}

}

for(i=0; i<4; i++)

{ for(j=0; j<i+1; j++)

{ System.out.print(twoD[i][j] + " ");

}

System.out.println();

}

}

}

Output:

0

1. 2
2. 4 5
3. 7 8

We can create a three-dimensional array where first index specifies the number of tables, second one number o0f rows and the third number of columns.

* Demonstrate a three-dimensional array. class threeDMatrix

{ public static void main(String args[])

{ int threeD[][][] = new int[3][4][5]; int i, j, k;

for(i=0; i<3; i++)

for(j=0; j<4; j++)

for(k=0; k<5; k++)

threeD[i][j][k] = i \* j \* k;

for(i=0; i<3; i++)

{

for(j=0; j<4; j++)

{

for(k=0; k<5; k++)

System.out.print(threeD[i][j][k] + " ");

System.out.println();

}

System.out.println();

}

}

}

Output:

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 0 0 0 0

0 1 2 3 4

0 2 4 6 8

0 3 6 9 12

0 0 0 0 0

0 2 4 6 8

0 4 8 12 16

0 6 12 18 24

Alternative Array Declaration Syntax

There is a second form that may be used to declare an array:

type[ ] var-name;

Here, the square brackets follow the type specifier, and not the name of the array variable. For example, the following two declarations are equivalent:

int al[] = new int[3];

int[] a2 = new int[3];

The following declarations are also equivalent:

char twod1[][] = new char[3][4];

char[][] twod2 = new char[3][4];

**Arithmetic Operators**

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Result | Operator | Result |
| + | Addition | -= | Subtraction Assignment |
| - | Subtraction | += | Addition Assignment |
| \* | Multiplication | \*= | Multiplication |
| / | Division | /= | Division Assignment |
| % | Modulus | %= | Modulus Assignment |
| ++ | Increment | -- | Decrement |

The operands of the arithmetic operators must be of a numeric type. You cannot use them on boolean types, but you can use them on char types, since the char type in Java is, essentially, a subset of int.

**The Basic Arithmetic Operators**

class BasicMath {

public static void main(String args[]) { // arithmetic using integers System.out.println("Integer Arithmetic"); int a = 1 + 1;

int b = a \* 3; int c = b / 4;

int d = c - a;

int e = -d;

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("c = " + c);

System.out.println("d = " + d);

System.out.println("e = " + e); // arithmetic using doubles

System.out.println("\nFloating Point Arithmetic");

double da = 1 + 1;

double db = da \* 3;

double dc = db / 4;

double dd = dc - a;

double de = -dd;

System.out.println("da = " + da);

System.out.println("db = " + db);

System.out.println("dc = " + dc);

System.out.println("dd = " + dd);

System.out.println("de = " + de); } }

When you run this program, you will see the following output:

Integer Arithmetic a = 2 b = 6 c = 1 d = -1 e = 1

Floating Point Arithmetic da = 2.0 db = 6.0 dc = 1.5 dd = -0.5 de = 0.5

**The Modulus Operator**

The my applied to floating-point types as well as integer types.

The following example program demonstrates the %:

//Demonstrate the % operator.

class Modulu

{ public static void main(String args[]))

{ int x = 42; double y = 42.25;

System.out.print("x mod 10 = " + x% 10);

System.out.println("y mod 10 = " + y % 10);

}

}

When you run this program, you will get the following output: x mod 10 = 2

y mod 10 = 2.25

Arithmetic Compound Assignment Operators Java provides special operators that can be used to combine an arithmetic operation with an assignment. As you probably know, statements like the following are quite common in programming: a = a + 4; In Java, you can rewrite this statement as shown here: a += 4;

This version uses the += compound assignment operator. Both statements perform the same action: they increase the value of a by 4. Here is another example, a = a % 2; which can be expressed as a %= 2; In this case, the %= obtains the remainder of a/2 and puts that result back into a.

There are compound assignment operators for all of the arithmetic, binary operators. Thus, any statement of the form var = var op expression; 60 Part I: The Java Language can be rewritten as var op= expression; The compound assignment operators provide two benefits.

First, they save you a bit of typing, because they are “shorthand” for their equivalent long forms. Second, they are implemented more efficiently by the Java run-time system than are their equivalent long forms. For these reasons, you will often see the compound assignment operators used in professionally written Java programs. Here is a sample program that shows several op= assignments in action: // Demonstrate several assignment operators.

class OpEquals

{

public static void main(String args[])

{ int a = 1; int b = 2; int c = 3;

a += 5; b \*= 4;

c += a \* b; c %= 6;

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("c = " + c);

}

}

The output of this program is shown here: a = 6 b = 8 c = 3

**Increment and Decrement**

The ++ and the – – are Java’s increment and decrement operators.

The increment operator increases its operand by one. The decrement operator decreases its operand by one.

For example, this statement: x = x + 1; can be rewritten like this by

use of the increment operator: x++;

Similarly, this statement: x = x - 1; is equivalent to x--;

The following program demonstrates the increment operator. // Demonstrate ++.

class IncDec

{ public static void main(String args[])

{ int a = 1; int b = 2; int c;

int d; c = ++b;

d = a++; c++;

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("c = " + c);

System.out.println("d = " + d);

}

}

The output of this program follows:

a = 2 b = 3 c = 4 d = 1

**The Bitwise Operators**

Java defines several bitwise operators that can be applied to the integer types, long, int, short, char, and byte. These operators act upon the individual bits of their operands. They are summarized in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Result** | **Operator** | **Result** |
| ~ | Bitwise unary NOT | &= | Shift left |
| & | Bitwise AND | != | Bitwise AND assignment |
| | | Bitwise OR | ^= | Bitwise OR assignment |
| ^ | Bitwise exclusive OR | >>= | Bitwise exclusive OR |
| >> | Shift right | >>>= | Shift right assignment |
| << | Shift left | <<= | Shift left assignment |
| >>> | Shift right zero fill |  |  |

All of the integer types are represented by binary numbers of varying bit widths. For example, the byte value for 42 in binary is 00101010, where each position represents a power of two, starting with 20 at the rightmost bit. The next bit position to the left would be 21 , or 2, continuing toward the left with 22 , or 4, then 8, 16, 32, and so on. So 42 has 1 bits set at positions 1, 3, and 5 (counting from 0 at the right); thus, 42 is the sum of 21 + 23 + 25 , which is 2 + 8 + 32.

All of the integer types (except char) are signed integers. This means that they can represent negative values as well as positive ones. Java uses an encoding known as two’s complement, which means that negative numbers are represented by inverting (changing 1’s to 0’s and vice versa) all of the bits in a value, then adding 1 to the result.

**The Bitwise Logical Operators**

The bitwise logical operators are &, |, ^, and ~. The following table shows the outcome of each operation. In the discussion that follows, keep in mind that the bitwise operators are applied to each individual bit within each operand.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **A** | **B** | **A|B** | **A&B** | **A^B** | **~A** |
|  |  |  |  |  |  |
| **0** | **0** | **0** | **0** | **0** | **1** |
|  |  |  |  |  |  |
| **1** | **0** | **1** | **0** | **1** | **0** |
|  |  |  |  |  |  |
| **0** | **1** | **1** | **0** | **1** | **1** |
|  |  |  |  |  |  |
| **1** | **1** | **1** | **1** | **0** | **0** |
|  |  |  |  |  |  |

The Bitwise NOT Also called the bitwise complement, the unary NOT operator, ~, inverts all of the bits of its operand. For example, the number 42, which has the following bit pattern: 00101010 becomes 11010101 after the NOT operator is applied.

The Bitwise AND The AND operator, &, produces a 1 bit if both operands are also 1. A zero is produced in all other cases.

Here is an example:

00101010 42

& 00001111 15

--------------------------------------------------

00001010 10

The Bitwise OR The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

00101010 42

| 00001111 15

---------------------------------------------------

00101111 47

The Bitwise XOR The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero.

The following example shows the effect of the ^.

This example also demonstrates a useful attribute of the XOR operation. Notice how the bit pattern of 42 is inverted wherever the second operand has a 1 bit. Wherever the second operand has a 0 bit, the first operand is unchanged. You will find this property useful when performing some types of bit manipulations.

00101010 42

* 00001111 15

--------------------------------------------

00100101 37

Using the Bitwise Logical Operators The following program demonstrates the bitwise logical operators: // Demonstrate the bitwise logical operators.

class BitLogic

{ public static void main(String args[])

{

String binary[] = { "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111", "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111" };

int a = 3; // 0 + 2 + 1 or 0011 in binary

int b = 6; // 4 + 2 + 0 or 0110 in binary

int c = a | b; int d = a & b;

int e = a ^ b;

int f = (~a & b) | (a & ~b);

int g = ~a & 0x0f;

System.out.println(" a = " + binary[a]);

System.out.println(" b = " + binary[b]);

System.out.println(" a|b = " + binary[c]);

System.out.println(" a&b = " + binary[d]);

System.out.println(" a^b = " + binary[e]);

System.out.println("~a&b|a&~b = " + binary[f]);

System.out.println(" ~a = " + binary[g]);

}

}

In this example, a and b have bit patterns that present all four possibilities for two binary digits: 0-0, 0-1, 1-0, and 1-1. You can see how the | and & operate on each bit by the results in c and d. The values assigned to e and f are the same and illustrate how the ^ works. The string array named binary holds the human-readable, binary representation of the numbers 0 through 15. In this example, the array is indexed to show the binary representation of each result. The array is constructed such that the correct string representation of a binary value n is stored in binary[n]. The value of ~a is ANDed with

0x0f (0000 1111 in binary) in order to reduce its value to less than 16, so it can be printed by use of the binary array.

Here is the output from this program:

a = 0011

b = 0110

a|b = 0111

a&b = 0010

a^b = 0101

~a&b|a&~b = 0101

~a = 1100

**The Left Shift**

The Left Shift The left shift operator, <<< num Here, num specifies the number of positions to left-shift the value in value. That is, the << moves all of the bits in the specified value to the left by the number of bit positions specified by num. For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right. This means that when a left shift is applied to an int operand, bits are lost once they are shifted past bit position 31. If the operand is a long, then bits are lost after bit position 63.

Java’s automatic type promotions produce unexpected results when you are shifting byte and short values. As you know, byte and short values are promoted to int when an expression is evaluated. Furthermore, the result of such an expression is also an int. This means that the outcome of a left shift on a byte or short value will be an int, and the bits shifted left will not be lost until they shift past bit position 31.

class MultByTwo

{

public static void main(String args[])

{ int i; int num = 0xFFFFFFE;

for(i=0; i<4;i++)

{ num=num<< 1; System.out.println(num);

}

}

}

The program generates the following output:

536870908

1073741816

2147483632

-32

**The Right Shift**

The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times. Its general form is shown here: value >> num 66

Here, num specifies the number of positions to right-shift the value in value. That is, the >> moves all of the bits in the specified value to the right the number of bit positions specified by num.

The following code fragment shifts the value 32 to the right by two positions, resulting in a being set to 8:

int a = 32;

a = a >> 2; // a now contains 8

When a value has bits that are “shifted off,” those bits are lost. For example, the next code fragment shifts the value 35 to the right two

positions, which causes the two low-order bits to be lost, resulting again in a being set to 8.

int a = 35;

a = a >> 2; // a still contains 8

Looking at the same operation in binary shows more clearly how this happens:

00100011 35

* 2

00001000 8

Each time you shift a value to the right, it divides that value by two— and discards any remainder. You can take advantage of this for high-performance integer division by 2. Of course, you must be sure that you are not shifting any bits off the right end.

**The Unsigned Right Shift**

If you are shifting something that does not represent a numeric value, you may not want sign extension to take place. This situation is common when you are working with pixel-based values and graphics. In these cases, you will generally want to shift a zero into the high-order bit no matter what its initial value was. This is known as an unsigned shift.

To accomplish this, you will use Java’s unsigned, shift-right operator, >>>, which always shifts zeros into the high-order bit. The following code fragment demonstrates the >>>.

Here, a is set to –1, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets a to 255. int a = -1; a = a >>> 24;

Here is the same operation in binary form to further illustrate what is happening:

11111111 11111111 11111111 11111111 –1 in binary as an int

>>>24

00000000 00000000 00000000 11111111 255 in binary as an int The

* operator is often not as useful as you might like, since it is only meaningful for 32- and 64-bit values. Remember, smaller values are automatically promoted to int in expressions. This means that sign-extension occurs and that the shift will take place on a 32-bit rather than on an 8- or 16-bit value.

**Bitwise Operator Compound Assignments**

All of the binary bitwise operators have a compound form similar to that of the algebraic operators, which combines the assignment with the bitwise operation.

For example, the following two statements, which shift the value in a right by four bits, are equivalent:

a = a >> 4;

a >>= 4;

Likewise, the following two statements, which result in a being assigned the bitwise expression a OR b, are equivalent: a = a | b; a |= b; The following program creates a few integer variables and then uses compound bitwise operator assignments to manipulate the variables:

class OpBitEquals

{ public static void main(String args[])

{ int a = 1; int b = 2;

int c = 3; a |= 4; b >>= 1;

c <<= 1; a ^= c;

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("c = " + c);

}

}

The output of this program is shown here: a = 3 b = 1 c = 6

**Relational Operators**

The relational operators determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

|  |  |
| --- | --- |
| **Operator** | **Result** |
|  |  |
| == | Equal to |
|  |  |
| != | Not equal to |
|  |  |
| > | Greater than |
|  |  |
| < | Less than |
|  |  |
| >= | Greater than and equal to |
|  |  |
| <= | Less than and equal to |
|  |  |

The outcome of these operations is a boolean value. The relational operators are most frequently used in the expressions that control the if statement and the various loop statements.

**Boolean Logical Operators**

The Boolean logical operators shown here operate only on boolean operands. All of the binary logical operators combine two boolean values to form a resultant Boolean value.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Result** | **Operator** | **Result** |
| & | Logical AND | &= | AND assignment |
| | | Logical OR | != | OR assignment |
| ^ | Logical XOR | ^= | XOR assignment |
| || | Short-circuit OR | == | Equal to |
| && | Short-circuit AND | != | Not equal to |
| ! | Logical unary NOT | ?: | Ternary if-then-else |

Demonstrate the boolean logical operators.

class BoolLogic

{ public static void main(String args[])

{ boolean a = true; boolean b = false;

boolean c = a | b; boolean d = a & b;

boolean e = a ^ b; boolean f = (!a & b) | (a & !b);

boolean g = !a;

System.out.println(" a = " + a);

System.out.println(" b = " + b);

System.out.println(" a|b = " + c);

System.out.println(" a&b = " + d);

System.out.println(" a^b = " + e);

System.out.println("!a&b|a&!b = " + f);

System.out.println(" !a = " + g);

}

}

**Short-Circuit Logical Operators**

Java provides two interesting Boolean operators not found in many other computer languages. These are secondary versions of the Boolean AND and OR operators, and are known as short-circuit logical operators. As you can see from the preceding table, the OR operator results in true when A is true, no matter what B is. Similarly, the AND operator results in false when A is false, no matter what B is. If you use the || and && forms, rather than the | and & forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone.

This is very useful when the right-hand operand depends on the value of the left one in order to function properly

if (denom != 0 && num / denom > 10)

Since the short-circuit form of AND (&&) is used, there is no risk of causing a run-time exception when denom is zero.

**The Assignment Operator**

The assignment operator is the single equal sign, =. The assignment operator works in Java much as it does in any other computer language. It has this general form: var = expression;

Here, the type of var must be compatible with the type of expression. The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments.

For example, consider this fragment:

int x, y, z; x = y = z = 100; // set x, y, and z to 100

This fragment sets the variables x, y, and z to 100 using a single statement.

This works because the = is an operator that yields the value of the right-hand expression. Thus, the value of z = 100 is 100, which is then assigned to y, which in turn is assigned to x. Using a “chain of assignment” is an easy way to set a group of variables to a common value.

**The ? Operator**

Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements. This operator is the ?. It can seem somewhat confusing at first, but the ? can be used very effectively once mastered.

The ? has this general form: expression1 ? expression2 : expression3

Here, expression1 can be any expression that evaluates to a boolean value. If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated. The result of the ? operation is that of the expression evaluated. Both expression2 and expression3 are required to return the same type, which can’t be void.

Here is an example of the way that the ? is employed:

ratio = denom == 0 ? 0 : num / denom;

When Java evaluates this assignment expression, it first looks at the expression to the left of the question mark. If denom equals zero, then the expression between the question mark and the colon is evaluated and used as the value of the entire ? expression. If denom does not equal zero, then the expression after the colon is evaluated and used for the value of the entire ? expression. The result produced by the ? operator is then assigned to ratio.

**Control Statements**

**If :** The if statement is examined in detail here. The if statement is Java’s conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the if statement:

if (condition)

statement1;

else

statement2;

Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a boolean value. The else clause is optional. The if works like this: If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed.

In no case will both statements be executed. For example, consider the following: int a, b; // ...

if(a < b) a = 0;

else b = 0;

**Nested ifs :** A nested if is an if statement that is the target of another if or else. Nested ifs are very common in programming. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else.

Here is an example:

if(i == 10)

{ if(j < 20) a = b;

if(k > 100) c = d; // this if is

else a = c; // associated with this else

}

else a = d; // this else refers to if(i == 10)

As the comments indicate, the final else is not associated with if(j100) because it is the closest if within the same block.

**The if-else-if Ladder**

A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder. It looks like this:

if(condition)

statement;

else if(condition)

statement;

else if(condition) statement;

. . else statement;

The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. The final else acts as a default condition; that is, if all other conditional tests fail, then the else statement is performed. If there is no final else and all other conditions are false, then no action will take place.

**switch**

The switch statement is Java’s multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of if-else-if statements.

Here is the general form of a switch statement:

switch (expression) {

case value1: // statement sequence

break;

case value2: // statement sequence

break; . . . case valueN: // statement sequence break;

default: // default statement sequence

}

The expression must be of type byte, short, int, or char; each of the values specified in the case statements must be of a type compatible with the expression.

The switch statement works like this: The value of the expression is compared with each of the literal values in the case statements. If a match is found, the code sequence following that case statement is executed. If none of the constants matches the value of the expression, then the default statement is executed. However, the default statement is optional. If no case matches and no default is present, then no further action is taken.

class SampleSwitch

{ public static void main(String args[])

{ for(int i=0; i<6;i++)

switch(i) {

case 0: System.out.println("i is zero.");

break;

case 1: System.out.println("i is one.");

break;

case 2: System.out.println("i is two.");

break;

case 3: System.out.println("i is three.");

break;

default: System.out.println("i is greater than 3.");

}

}

}

The output produced by this program is shown here:

i is zero.

i is one.

i is two.

i is three.

i is greater than 3.

i is greater than 3.

**Nested switch Statements**

You can use a switch as part of the statement sequence of an outer switch. This is called a nested switch. Since a switch statement defines its own block, no conflicts arise between the case constants in the inner switch and those in the outer switch.

For example, the following fragment is perfectly valid:

switch(count) {

case 1: switch(target) { // nested switch

case 0: System.out.println("target is zero");

break; case 1: // no conflicts with outer switch

System.out.println("target is one"); break; } break; case 2: // ... Here,

the case 1: statement in the inner switch does not conflict with the case 1: statement in the outer switch. The count variable is only compared with the list of cases at the outer level. If count is 1, then target is compared with the inner list cases.

In summary, there are three important features of the switch statement to note:

* The switch differs from the if in that switch can only test for equality, whereas if can evaluate any type of Boolean expression. That is, the switch looks only for a match between the value of the expression and one of its case constants.
* No two case constants in the same switch can have identical values. Of course, a switch statement and an enclosing outer switch can have case constants in common.
* A switch statement is usually more efficient than a set of nested ifs.

**Iteration Statements**

Java’s iteration statements are for, while, and do-while. These statements create what we commonly call loops.

while The while loop is Java’s most fundamental loop statement. It repeats a statement or block while its controlling expression is true. Here is its general form: while(condition) { // body of loop } The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

class While

{ public static void main(String args[])

{ int n = 10;

while(n > 0)

{ System.out.println("tick " + n);

n--;

}

}

}

Since the while loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with. For example, in the following fragment, the call to println( ) is never executed:

int a = 10, b = 20;

while(a > b)

System.out.println("This will not be displayed");

**do-while :** As you just saw, if the conditional expression controlling a while loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with.

Java supplies a loop that does just that: the do-while. The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

do {

// body of loop

} while (condition);

Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java’s loops, condition must be a Boolean expression.

// Demonstrate the do-while loop.

class DoWhile

{ public static void main(String args[])

{ int n = 10;

do {

System.out.println("tick " + n);

n--;

} while(n > 0);

}

}

The do-while loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once.

Consider the following program, which implements a very simple help system for Java’s selection and iteration statements: // Using a do-while to process a menu selection.

import java.util.Scanner;  
class Menu   
{ public static void main(String args[]) throws java.io.IOException  
 { Scanner in=new Scanner(System.in);  
 int choice;  
 do   
 { System.out.println("Help on:");   
 System.out.println(" 1. if");   
 System.out.println(" 2. switch");   
 System.out.println(" 3. while");   
 System.out.println(" 4.do-while");   
 System.out.println(" 5. for\n");   
 System.out.println("Enter Choice from 1 to 5:");  
 choice=in.nextInt();   
 switch(choice)  
 {  
 case 1 : System.out.println("The if:\n");  
 System.out.println("if(condition) statement;");  
 System.out.println("else statement;");  
 break;  
 case 2 : System.out.println("The switch:\n");  
 System.out.println("switch(expression) {");  
 System.out.println(" case constant:");  
 System.out.println(" statement sequence");  
 System.out.println(" break;");  
 System.out.println(" // ...");  
 System.out.println("}");  
 break;  
 case 3: System.out.println("The while:\n");  
 System.out.println("while(condition) statement;");  
 break;  
 case 4 : System.out.println("The do-while:\n");  
 System.out.println("do {");  
 System.out.println(" statement;");  
 System.out.println("}while (condition);");   
 break;  
 case 5: System.out.println("The for:\n");  
 System.out.print("for(init; condition;iteration)");  
 System.out.println(" statement;");  
 break;  
 default : System.out.println("Choice out of range;");  
 }  
 }while( choice >=1 && choice <=5);   
   
 }  
}

**for**

The first is the traditional form that has been in use since the original version of Java. The second is the new “for-each” form.

Here is the general form of the traditional for statement: for(initialization; condition; iteration) { // body } If only one statement is being repeated, there is no need for the curly braces.

* The for loop operates as follows. When the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once.
* Next, condition is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.
* Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

class ForTick

{

public static void main(String args[])

{ int n;

for(n=10; n>0; n--)

System.out.println("tick " + n);

}

}

**Declaring Loop Control Variables Inside the for Loop**

* Declare a loop control variable inside the for. class ForTick {

public static void main(String args[]) {

//here, n is declared inside of the for loop

for(int n=10; n>0; n--)

System.out.println("tick " + n);

}

}

When you declare a variable inside a for loop, there is one important point to remember: the scope of that variable ends when the for statement does. (That is, the scope of the variable is limited to the for loop.) Outside the for loop, the variable will cease to exist.

**Using the Comma**

Using the comma, the preceding for loop can be more efficiently coded as shown here: // Using the comma.

// multiple varaible inside the for loop

class MultipleVariableForLoop

{ public static void main(String args[])

{ int a, b;

for(a=1, b=7; a<b;a=a+2,b++)

{

System.out.println("a = " + a);

System.out.println("b = " + b);

}

}

}

Output is :

a = 1 b=7

a = 3 b=8

a = 5 b=9

a = 7 b=10

a = 9 b=11

a = 11 b=12

**Some for Loop Variations**

One of the most common variations involves the conditional expression. Specifically, this expression does not need to test the loop control variable against some target value.

In fact, the condition controlling the for can be any Boolean expression. For example, consider the following fragment:

boolean done = false;

for(int i=1; !done; i++) {

// ...

if(interrupted()) done = true; }

In this example, the for loop continues to run until the boolean variable done is set to true. It does not test the value of i.

**The For-Each Version of the for Loop**

The general form of the for-each version of the for is shown here:

for(type itr-var : collection) statement-block

Here, type specifies the type and itr-var specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by collection. With each iteration of the loop, the next

element in the collection is retrieved and stored in itr-var. The loop repeats until all elements in the collection have been obtained.

int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int sum = 0;

for(int x: nums)

sum += x;

**Iterating Over Multidimensional Arrays**

The enhanced version of the for also works on multidimensional arrays. Remember, however, that in Java, multidimensional arrays consist of arrays of arrays. This is important when iterating over a multidimensional array, because each iteration obtains the next array, not an individual element. Furthermore, the iteration variable in the for loop must be compatible with the type of array being obtained.

class ForEach3   
{ public static void main(String args[])   
 { int sum = 0;  
 int nums[][] = new int[3][5]; // give nums some values  
 for(int i = 0; i < 3; i++)  
 for(int j=0; j < 5; j++)  
 nums[i][j] = (i+1)\*(j+1);   
 // use for-each for to display and sum the values  
 for(int x[] : nums)   
 { for(int y : x)  
 { System.out.println("Value is: " + y);  
 sum += y;  
 }  
 }  
 System.out.println("Summation: " + sum);   
 }  
 }

for(int x[] : nums)

Notice how x is declared. It is a reference to a one-dimensional array of integers. This is necessary because each iteration of the for obtains the next array in nums, beginning with the array specified by nums[0]. The inner for loop then cycles through each of these arrays, displaying the values of each element.

**Applying the Enhanced for**

class Search

{

public static void main(String args[])

{ int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };

int val = 5;

boolean found = false;

// use for-each style for to search nums for val

for(int x : nums) {

if(x == val) {

found = true; break;

}

}

if(found)

System.out.println("Value found!");

}

}

The for-each style for is an excellent choice in this application because searching an unsorted array involves examining each element in sequence.

Other types of applications that benefit from for-each style loops include computing an average, finding the minimum or maximum of a set, looking for duplicates, and so on.

**Nested Loops**

Java allows loops to be nested. That is, one loop may be inside another.

* Loops may be nested.

class Nested

{

public static void main(String args[])

{ int i, j;

for(i=0; i<10; i++) {

for(j=i; j<10; j++)

System.out.print(".");

System.out.println();

}

}

**Jump Statements**

Java supports three jump statements: break, continue, and return.

These statements transfer control to another part of your program.

**Using break**

By using break, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the  body of the loop. When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. Here is a simple example:

// Using break to exit a loop.

public class BreakLoop

{ public static void main(String args[])

{ for(int i=0; i<100; i++)

{ if(i == 10)

break; // terminate loop if i is 10 System.out.println("i: " + i);

}

System.out.println("Loop complete.");

}

}

The break statement can be used with any of Java’s loops, including intentionally

infinite loops. For example, here is the preceding program coded by use of a while loop.

The output from this program is the same as just shown.

//Using break to exit a while loop.

class BreakLoop2

{ public static void main(String args[])

{ int i = 0;

while(i < 100)

{

if(i == 10) break; // terminate loop if i is 10

System.out.println("i: " + i);

i++;

}

System.out.println("Loop complete.");

}

}

**Using break as a Form of Goto**

In addition to its uses with the switch statement and loops, the break statement can also be employed by itself to provide a “civilized” form of the goto statement. Java does not have a goto statement because it provides a way to branch in an arbitrary and unstructured manner. This usually makes goto-ridden code hard to understand and hard to maintain. It also prohibits certain compiler optimizations.

For example, the goto can be useful when you are exiting from a deeply nested set of loops. To handle such situations, Java defines an expanded form of the break statement. By using this form of break, you can, for example, break out of one or more blocks of code. These blocks need not be part of a loop or a switch. They can be any block. Further, you can specify precisely where execution will resume, because this form of break works with a label.

The general form of the labeled break statement is shown here:

break label;

Most often, label is the name of a label that identifies a block of code. This can be a stand-alone block of code but it can also be a block that is the target of another statement. When this form of break executes, control is transferred out of the named block.

The labeled block must enclose the break statement, but it does not need to be the immediately enclosing block. This means, for example, that you can use a labeled break statement to exit from a set of nested blocks.

But you cannot use break to transfer control out of a block that does not enclose the break statement.

To name a block, put a label at the start of it. A label is any valid Java identifier followed by a colon. Once you have labeled a block, you can then use this label as the target of a break statement.

class Break

{ public static void main(String args[])

{ boolean t = true;

first: {

second: {

third: {

System.out.println("Before the break.");

if(t) break second; // break out of second block

System.out.println("This won't execute");

}

System.out.println("This won't execute");

}

System.out.println("This is after second block.");

}

}

}

**Using continue**

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop’s end. The continue statement performs such an action. In while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop. In a for loop, control goes first to the iteration portion of the for statement and then to the conditional expression.

Here is an example program that uses continue to cause two numbers to be printed on each line:

//Demonstrate continue.

class Continue

{ public static void main(String args[])

{ for(int i=0; i<10; i++)

{ System.out.print(i + " ");

if (i%2 == 0) continue;

System.out.println("");

}

}

}

As with the break statement, continue may specify a label to describe which enclosing loop to continue. Here is an example program that

uses continue to print a triangular multiplication table for 0 through 9.

class ContinueLabel

{ public static void main(String args[])

{ outer: for (int i=0; i<10; i++)

{

for(int j=0; j<10; j++)

{

if(j > i)

{ System.out.println();

continue outer;

}

System.out.print(" " + (i \* j));

}

}

System.out.println();

}

}

**Return**

The last control statement is return. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement.

At any time in a method the return statement can be used to cause execution to branch back to the caller of the method. Thus, the return statement immediately terminates the method in which it is executed.

The following example illustrates this point. Here, return causes execution to return to the Java run-time system, since it is the run-time system that calls main( ).

class Return

{ public static void main(String args[])

{ boolean t = true;

System.out.println("Before the return.");

if(t) return; // return to caller

System.out.println("This won't execute.");

}

}