MODULE - 1

An Overview of Java: Object-Oriented Programming, A First Simple Program, A Second Short Program, Two Control Statements, Using Blocks of Code, Lexical Issues, The Java Class Libraries, Data Types, Variables, and Arrays: Java Is a Strongly Typed Language, The Primitive Types, Integers, Floating-Point Types, Characters, Booleans, A Closer Look at Literals, Variables, Type Conversion and Casting, Automatic Type Promotion in Expressions, Arrays, A Few Words About Strings Text book 1: Ch 2, Ch 3

➢ Java Installation-Environment Setup

- Java SE is freely available from the link Download Java. You can download a version based on your operating system.
- Follow the instructions to download Java and run the .exe to install Java on your machine. Once you installed Java on your machine, you will need to set environment variables to point to correct installation directories.

➢ Setting Up the Path for Windows

- Assuming you have installed Java in c:\Program Files\java\jdk directory
- Right-click on 'My Computer' and select 'Properties'.
- Click the 'Environment variables' button under the 'Advanced' tab.
- Now, alter the 'Path' variable so that it also contains the path to the Java executable. Example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'.

➢ Popular Java Editors

To write your Java programs, you will need a text editor. There are even more sophisticated IDEs available in the market. But for now, you can consider one of the following –

- Notepad – On Windows machine, you can use any simple text editor like Notepad (Recommended for this tutorial), TextPad.
- Netbeans – A Java IDE that is open-source and free which can be downloaded from https://www.netbeans.org/index.html.
- Eclipse – A Java IDE developed by the eclipse open-source community and can be

downloaded from https://www.eclipse.org/.

➢ 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. OOP is so integral to Java that it is best to understand its basic principles before you begin writing even simple Java programs.

Two Paradigms

- All computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data.
- The first way is called the *process-oriented model.* This approach characterizes a program as a series of linear steps (that is, code). The process-oriented model can be thought of as *code acting on data.* Procedural languages such as C employ this model to considerable success.
- To manage increasing complexity, the second approach, called *object-oriented programming,* was conceived. Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as *data controlling access to code.*

Abstraction

- Process of identifying the essential details to be known and ignoring the non-essential details from the perspective of the end users.
- An essential element of object-oriented programming is *abstraction.* Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of 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 to drive to the grocery store without being overwhelmed by the complexity of the parts that form the car. They can ignore the details of how the engine, transmission, and braking systems work. Instead, they are free to utilize the object as a whole.

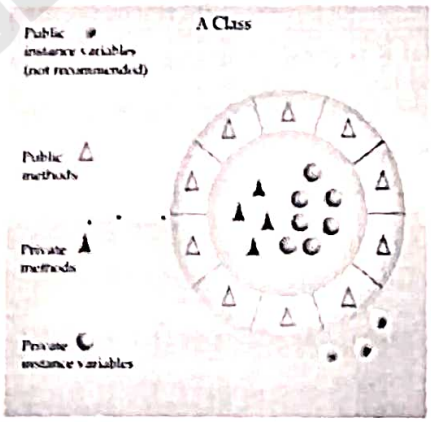
➢ The Three OOP Principles

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

1. 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.
- Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked **private or public.**
- The *public* interface of a class represents everything that external users of the class need to know, or may know.
- The *private* methods and data can only be accessed by code that is a member of the class. Therefore, any other code that is not a member of the class cannot access a private method or variable.

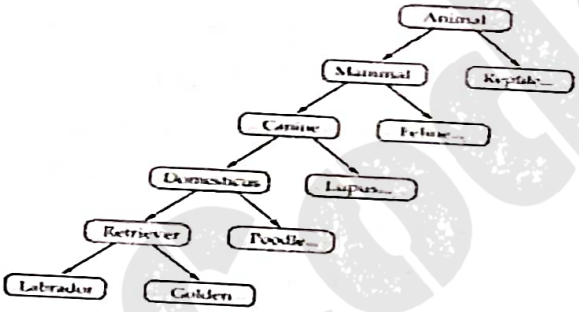
FIGURE 3-1 Encapsulation: public methods can be used to protect private data



2. Inheritance

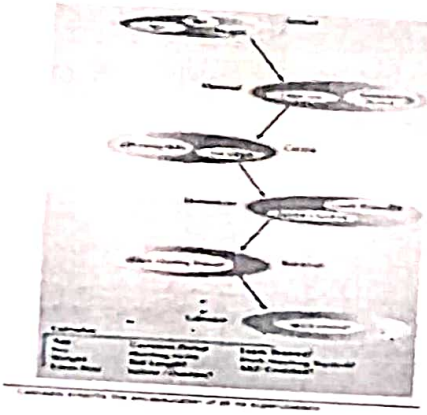
- *Inheritance* is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification.

- For example, a Golden Retriever is part of the classification dog, which in turn is part of the mammal class, which is under the larger class animal.
- Without the use of hierarchies, each object would need to define all of its characteristics explicitly. By use of inheritance, an object need only define those qualities that make it unique within its class.
- It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.



3. Polymorphism

- *Polymorphism* (from Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. 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.*



➢ A First Simple Program:

```
/* This is a simple Java program.
Call this file "Example.java". */
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.");
} }
```

1. **Entering the Program:**

   The first thing that you must learn about Java is that the name you give to a source file is very important. For this program the name of the source file should be **Example.java**.

2. **Compiling the Program:**

   To compile the **Example** program, execute the compiler, **javac**, specifying the name of the source file on the command line, as shown here:

   C:\>javac Example.java

   - The **javac** compiler creates a file called **Example.class** that contains the bytecode version of the program.
   - The Java bytecode is the intermediate representation of your program that contains instructions the Java Virtual Machine will execute. Thus, the output of **javac** is not code that can be directly executed. To 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.

     Output: This is a simple Java program.

   - **public static void main(String args[])** // Your program begins with a call to main(). This line begins the **main( )** method at which the program will begin executing by calling **main()**.
   - The **public** keyword is an *access specifier*, member may be accessed by code outside the class in which it is declared. (The opposite of **public** is **private**, which prevents a

member from being used by code defined outside of its class.)

- The keyword **static** allows **main( )** to be called without having to instantiate a particular instance of the class. This is necessary since **main( )** is called by the Java Virtual Machine before any objects are made. The keyword **void** simply tells the compiler that **main( )** does not return a value.
- In **main( ), String args[ ]** declares a parameter named **args**, which is an array of instances of the class **String**. (*Arrays* are collections of similar objects.) Objects of type **String** store character strings. In this case, **args** receives any command-line arguments present when the program is executed.

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 accomplished by the built-in **println( )** method. **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console.

➢ Comments:

Comment	Description
/* text */ Multi-line comment	The compiler ignores everything from /* to */.
//text Single Line comment	The compiler ignores everything from // to the end of the line.
/**documentation*/ Document Comment	This is a documentation comment and in general its called doc comment. The JDK java doc tool uses doc comments when preparing automatically generated documentation.

➢ A Second Short Program:

```
/*
   Here is another short example.
   Call this file "Example2.java".
*/

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:

```
This is num: 100
The value of num * 2 is 200
```

- The first new line in the program is shown here:

  int num; // this declares a variable called num.

- This line declares an integer variable called num. Java (like most other languages) requires that variables be declared before they are used. Following is the general form of a variable declaration:

  type var-name

- Here, ***type*** specifies the type of variable being declared, and ***var-name*** is the name of the variable. If you want to declare more than one variable of the specified type, you may use a comma-separated list of variable names. The keyword int specifies an integer type.
- In the program, **the line num = 100;** // this assigns num the value 100 assigns to num the value 100. In Java, the assignment operator is a single equal sign.
- The next line of code outputs the value of num preceded by the string "This is num:"

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

- In this statement, the plus sign causes the value of num to be appended to the string that precedes it, and then the resulting string is output. Using the + operator, you can join together as many items as you want within a single **println( )** statement. The next line of code assigns num the value of num times 2.

- Java uses the * operator to indicate multiplication. After this line executes, num will contain the value 200. Here are the next two lines in the program:

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

System.out.println(num);

- This string is not followed by a newline. This means that when the next output is generated, it will start on the same line. The **print( )** method is just like **println( )**, except that it does not output a newline character after each call.

➢ Two Control Statements

- **The if Statement:** The Java if statement works much like the IF statement in any other language. Further, it is syntactically identical to the if statements in C, C++, and C#. Its simplest form is shown here.

if(condition) statement;

- Here, condition is a Boolean expression. If condition is true, then the statement is executed. If condition is false, then the statement is bypassed. Here is an example:

if(num < 100)

System.out.println("num is less than 100");

- If num contains a value that is less than 100, the conditional expression is true, and println( ) will execute. If num contains a value greater than or equal to 100, then the println( ) method is bypassed.
- Many relational operators can be used in the if condition

Operator	Meaning
<	Less than
>	Greater Than
==	Equal to

For example,

```
/*
  Demonstrate the if.

  Call this file "IfSample.java".
*/
class IfSample {
  public static void main(String args[]) {
    int x, y;

    x = 10;
    y = 20;

    if(x < y) System.out.println("x is less than y");

    x = x * 2;
    if(x == y) System.out.println("x now equal to y");

    x = x * 2;
    if(x > y) System.out.println("x now greater than y");

    // this won't display anything
    if(x == y) System.out.println("you won't see this");
  }
}
```

The output generated by this program is shown here:

```
x is less than y
x now equal to y
x now greater than y
```

Notice one other thing in this program. The line

```
int x, y;
```

declares two variables, x and y, by use of a comma-separated list.

➢ The for Loop:

- The simplest form of the for loop is shown here:

  for(initialization; condition; iteration) statement;

- The initialization portion of the loop sets a loop control variable to an initial value. The condition is a Boolean expression that tests the loop control variable. If the outcome of that test is true, the for loop continues to iterate. If it is false, the loop terminates. The iteration expression determines how the loop control variable is changed each time the loop iterates. Here is a short program that illustrates the for loop:

```
/*
  Demonstrate the for loop.

  Call this file "ForTest.java".
*/
class ForTest {
  public static void main(String args[]) {
    int x;

    for(x = 0; x<10; x = x+1)
      System.out.println("This is x: " + x);
  }
}
```

This program generates the following output:

```
This is x: 0
This is x: 1
This is x: 2
This is x: 3
This is x: 4
This is x: 5
This is x: 6
This is x: 7
This is x: 8
This is x: 9
```

- In this example, x ~~is the loop control variable. It is initialized to zero in the initialization~~ portion of the for. At the start of each iteration (including the first one), the conditional test x < 10 is performed. If the outcome of this test is true, the println( ) statement is executed, and then the iteration portion of the loop is executed. This process continues until the conditional test is false.

➢ Using Blocks of Code

- Java allows two or more statements to be grouped into blocks of code, also called code blocks. This is done by enclosing the statements between opening and closing curly braces. Once, a block of code has been created, it becomes a logical unit that can be used any place that a single statement can.

```
if(x < y) { // begin a block
  x = y;
  y = 0;
} // end of block
```

- Consider this if statement: Here, if x is less than y, then both statements inside the block

will be executed. Thus, the two statements inside the block form a logical unit, and one statement cannot execute without the other also executing. The key point here is that whenever you need to logically link two or more statements, you do so by creating a block.

➢ Lexical Issues

Java programs are a collection of whitespace, identifiers, literals, comments, operators, separators, and keywords.

- **Whitespace:** In Java, whitespace is a space, tab, or newline.
- **Identifiers:** Identifiers are used for class names, method names, and variable names. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. Java is case-sensitive, so **VALUE** is a different identifier than **Value**. In Java, there are several points to remember about identifiers. They are as follows –
    1. All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
    2. After the first character, identifiers can have any combination of characters.
    3. A key word cannot be used as an identifier. Identifiers are case sensitive.
    4. Examples of legal identifiers: age, $salary, _value, l_value.
    5. Examples of illegal identifiers: 123abc, -salary]
    6. Some valid identifiers are:

AvgTemp	count a4	$test	this_is_ok

    7. Invalid identifiers are:

count	high-temp	Not/ok

- **Literals:** A constant value in Java is created by using a literal representation of it. For example, here are some literals: 100 98.6 'X' "This is a test". Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string.
- **Comments:** There are three types of comments defined by Java. You have already seen two: single-line and multiline. The third type is called a *documentation comment.* This

type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a /** and ends with a * /.

- **Separators:** In Java, there are a few characters that are used as separators. The most used separator in Java is the semicolon. As you have seen, it is used to terminate statements. The separators are shown in the following table:

Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a for statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.

➢ The Java Keywords

- There are 50 keywords currently defined in the Java language. These keywords, combined with the syntax of the operators and separators, form the foundation of the Java language. These keywords cannot be used as names for a variable, class, or method. The keywords const and goto are reserved but not used.

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

➢ The Java Class Libraries

- The Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking, and graphics. The standard classes also provide support for windowed output. Thus, Java

as a totality is a combination of the Java language itself, plus its standard classes. Example: Java's built-in methods: println( ) and print() are members of the System class.

➢ Java Is a Strongly Typed Language

It is important to state at the outset that Java is a strongly typed language.

- Every variable has a type, every expression has a type, and every type is strictly defined.
- All assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
- There are no automatic coercions or conversions of conflicting types as in some languages.
- The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

➢ Primitive Data 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.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.33.
- **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.

a) Integers

- Java defines four integer types: byte, short, int, and long. All of these are signed, positive and negative values.

- The width of an integer type should not be thought of as the amount of storage it consumes, but rather as the behavior it defines for variables and expressions of that type.
- The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them.

Name	Width	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	-128 to 127

b) 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.
- Byte variables are declared by use of the byte keyword. For example, the following declares two byte variables called b and c:

byte b, c

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;

d) int

- 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.
- When byte and short values are used in an expression they are promoted to int when the expression is evaluated.

e) long

- long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value. The range of a **long** is quite large.

```
// Compute distance light travels using long variables.
        class Light {
        public static void main (String args[]) {
        int lightspeed;
        long days;
        long seconds;
        long distance;
        lightspeed = 186000; // approximate speed of light in miles per second
        days = 1000; // specify number of days here
        seconds = days * 24 * 60 * 60; // convert to seconds
        distance = lightspeed * seconds; // compute distance
        System.out.print("In " + days);
         System.out.print(" days light will travel about ");
         System.out.println(distance + " miles.");
         } }
```

Output:

This program generates the following output:

In 1000 days light will travel about 16070400000000 miles.

➢ 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 transcendental 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.

a) float

- The type float specifies a *single-precision* value that uses 32 bits of storage. For example, float can be useful when representing dollars and cents. Here are some example for float variable declarations: **float hightemp, lowtemp;**

b) 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. 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);
}
}
```

c) Characters

- In Java, the data type used to store characters is **char**.
- **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. 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.
- In Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536. There are no negative **chars**. Here is a program that demonstrates **char** variables:

```
// char variables behave like integers
class CharDemo {
public static void main(String args[]) {
char ch1;
ch1 = 'X';
System.out.print("character in ch1 is: " +ch1);
ch1++; //increment ch1
System.out.println("ch1 is now " + ch1);
}
}
```

Output: character in ch1 is: X

ch1 is now Y

- In the program, ch1 is first given the value X. Next, ch1 is incremented. This results in ch1 containing Y, the next character in the ASCII (and Unicode) sequence.

d) Boolean

- 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:

```
// Demonstrate boolean values.
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

- First, 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) .

- Third, the outcome of a relational operator, such as 9 displays the value "true."
- Further, the extra set of parentheses around 10 > 9 is necessary because the + operator has a higher precedence than the >.

Note: Study the different types of literals from textbook.

➢ 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.

a. Declaring a Variable

- 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 identifier is the name of the variable. You can initialize the variable by specifying an equal sign and a value. 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, initializing // d and f.
byte z = 22; // initializes z.
double pi = 3.14159; // declares an approximation of pi.
char x = 'x'; // the variable x has the value 'x'.
```

b. Dynamic Initialization

- 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);
} }
```

- There are three local variables—a, b, and c—are declared.
- The first two, a and b, are initialized by constants.
- c is initialized dynamically to the length of the hypotenuse.
- 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 Scope and Lifetime of Variables

- Java allows variables to be declared within any block. A block is begun with an opening curly brace and ended by a closing curly brace. A block defines a *scope.*
- A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.
- Many other computer languages define two general categories of scopes: global and local. These traditional scopes do not fit well with Java's strict, object-oriented model.
- As a general rule, 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.

```
// 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
// x and y both known here.
System.out.println("x and y: " + x + " " + y);
x = y * 2;
}
```

```
// y = 100; // Error! y not known here
// x is still known here.
System.out.println("x is " + x);
} }
```

- 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 is only 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.
- Within a block, variables can be declared at any point, but are valid only after they are declared. Thus, if you define a variable at the start of a method, it is available to all of the code within that method.
- If you declare a variable at the end of a block, it is effectively useless, because no code will have access to it.
- For example, this fragment is invalid because count cannot be used prior to its declaration:

```
// This fragment is wrong! count = 100;
// oops! cannot use count before it is declared! int count;
```

- 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
```

➢ Type Conversion and Casting

- The process of converting one data type to another data type is known as type conversion. 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.
- Not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from **double** to **byte**. Fortunately, it is still possible to obtain a 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 met:

- 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, there are no automatic conversions from the numeric types to **char** or **boolean**. Also, **char** and **boolean** are not compatible with each other.

➢ Casting Incompatible Types

- Although the automatic type conversions are helpful, they will not fulfill all needs.
- For example, if you 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 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. For example, the following fragment casts an **int** to a **byte**. If the integer's value is larger than the range of a **byte**, it will be reduced modulo (the remainder of an integer division by the) **byte**'s range.

```
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.
- Integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.

The following program demonstrates some type conversions that require casts:

```
// Demonstrate casts.
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("\n Conversion of double to int.");
i = (int) d;
System.out.println("d and i " + d + " " + i);
System.out.println("\n Conversion of double to byte.");
b = (byte) d;
System.out.println("d and b " + d + " " + b);
} }
```

This program generates the following 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
```

➢ Arrays

- An array is a group of like-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.

a) One-Dimensional Arrays: A one-dimensional array is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general formof a one-dimensional array declaration is

type var-name[ ];

- Here, type declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines

what type of data the array will hold. For example, the following declares an array named month_days with the type "array of int":

int month_days[]

- 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. month_days = **new** int[12]
- Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero.
- For example, this statement assigns the value 28 to the second element of month_days.
  month_days[1] = 28;
- The next line displays the value stored at index 3.
  System.out.println(month_days[3])

// Demonstrate a one-dimensional array.

```
class Array {
public static void main(String args[]) {
int month_days[];
month_days = new int[12];
month_days[0] = 31;
month_days[1] = 28;
month_days[2] = 31;
month_days[3] = 30;
month_days[4] = 31;
month_days[5] = 30;
month_days[6] = 31;
month_days[7] = 31;
month_days[8] = 30;
month_days[9] = 31;
```

```
month_days[10] = 30;
month_days[11] = 31;
System.out.println("April has " + month_days[3] + " days.");
} }
```

- When you run this program, it prints the number of days in April. Java array indexes start with zero, so the number of days in April is month_days[3] or 30.
- It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here: **int month_days[] = new int[12];**
- Arrays can be initialized when they are declared. An array initializer is a list of comma-separated expressions surrounded by curly braces.
- 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. Example below:

```
class AutoArray {
public static void main(String args[]) {
int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
System.out.println("April has " + month_days[3] + " days.");
} }
```

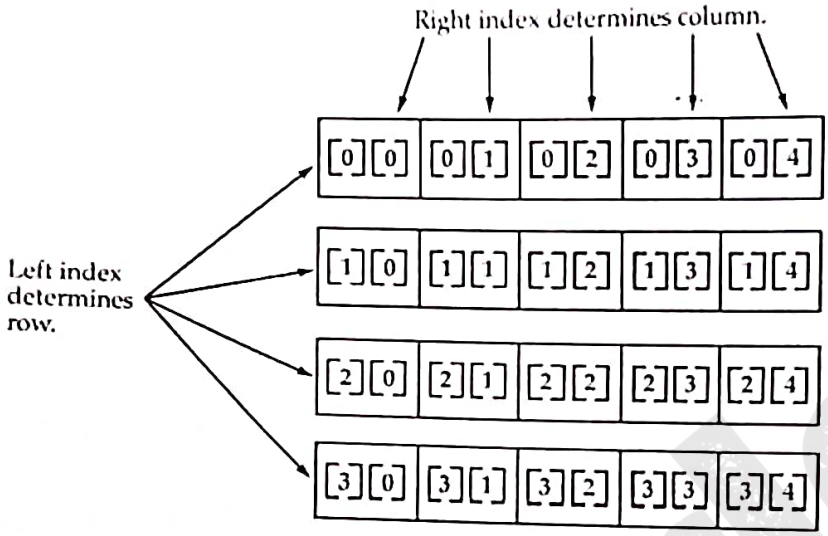
- Java strictly checks to make sure you do not accidentally try to store or reference values outside of the range of the array. The Java run-time system will check to be sure that all array indexes are in the correct range.

b) 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.



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

// Demonstrate a two-dimensional array.

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

This program generates the following 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 dimension

separately. For example, this following code allocates memory for the first dimension of twoD when it is declared. It 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];
```

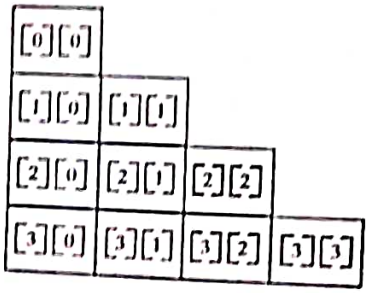
- While there is no advantage to individually allocating the second dimension arrays in this situation, there may be in others.

// 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(); }}}
```

This program generates the following output:

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



➢ 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];
- This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

  int[] nums, nums2, nums3; **// create three arrays**
- creates three array variables of type int. It is the same as writing

  int nums[], nums2[], nums3[]; **// create three arrays**

  The alternative declaration form is also useful when specifying an array as a return type for a method.

➢ A Few Words About Strings

- Java's string type, called String, is not a simple type nor is it simply an array of characters.
- A String defines an object, and a full description of it requires an understanding of several object- related features. The String type is used to declare string variables. You can also declare arrays of strings. A quoted string constant can be assigned to a String variable.
- A variable of type String can be assigned to another variable of type String. You can use an object of type String as an argument to println( ).

- For example, consider the following fragment:

  String str = "this is a test";

  System.out.println(str);

- Here, str is an object of type String. It is assigned the string "this is a test". This string is displayed by the println( ) statement.
- String objects have many special features and attributes that make them quite powerful and easy to use.

Operators: Arithmetic Operators, The Bitwise Operators, Relational Operators, Boolean Logical Operators, The Assignment Operators, The ? Operator, Operator Precedence, Using Parentheses, Control Statements: Java's Selection Statements, Iteration Statements, Jump Statements Text book 1: Ch 4, Ch 5

➢ Operators

Java provides rich set of operators, mainly divided into four groups viz. arithmetic, bitwise, relational and logical.

➢ 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	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
++	Increment
--	Decrement
+=	Addition assignment

-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment

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 a subset of int.

➢ The Basic Arithmetic Operator:

- Basic arithmetic operators like +, -, * and / behave as expected for numeric data. The – symbol can be used as unary operator to negate a variable.
- If / is operated on two integer operands, then we will get only integral part of the result by truncating the fractional part.
- The % operator returns the remainder after division. It can be applied on integer and floating-point types. The following simple example program demonstrates the arithmetic operators. It also illustrates the difference between floating-point division and integer division.

```
// Demonstrate 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 modulus operator, %, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types. The following example program demonstrates the %:

```
// Demonstrate the % operator.
class Modulus {
  public static void main(String args[]) {
    int x = 42;
    double y = 42.25;

    System.out.println("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. 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 can be rewritten as

var = var op expression;

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. 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
```

Note: Increment and decrement already discussed in module-1.

➢ Relational Operators

- The relational operators determine the relationship between two operands. Specifically, they determine equality and ordering among operands. Following table lists the relational operators supported by Java.

Operator	Meaning
==	Equal to (or comparison)
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

- The outcome of these operations is a **boolean** value. Any type in Java, including integers, floating-point numbers, characters, and Booleans can be compared using the equality test, ==, and the inequality test, !=.
- Only numeric types can be compared using the ordering operators. That is, only integer, floating- point, and character operands may be compared to see which is greater or less than the other. For example, the following code fragment is perfectly valid:

  int a = 4; int b = 1;

  boolean c = a < b;
- In this case, the result of **a<b** (which is **false**) is stored in **c**. **Note** that in C/C++ we can have following type of statement –

```
int done;
// ...
if(!done) ...    // Valid in C/C++
if(done) ...     // but not in Java.
```

In Java, these statements must be written like this:

```
if(done == 0) ... // This is Java-style.
if(done != 0) ...
```

➢ 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	Meaning
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

- The truth table is given below for few operations:

A	B	A|B	A&B	A^B	!A
False	False	False	False	False	True
False	True	True	False	True	True
True	False	True	False	True	False
True	True	True	True	False	False

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

- After running this program, you will see that the same logical rules apply to Boolean values as they did to bits. As you can see from the following output, the string representation

of a Java boolean value is one of the literal values true or false:

```
a = true
b = false
a|b = true
a&b = false
a^b = true
a&b|a&!b = true
!a = false
```

➢ The Assignment Operator

- The *assignment operator* is the single equal sign, =. It has this general form:

var = expression;

- Here, the type of *var* must be compatible with the type of *expression*. 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 supports *ternary operator* which sometimes can be used as an alternative for *if*-then-*else* statement. The general form is –

var = expression1 ? expression2 : expression3;

- Here, *expression1* is evaluated first and it must return Boolean type. If it results *true*, then value of *expression2* is assigned to *var*, otherwise value of *expression3* is assigned to *var*. For example,

```
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.

- Here is a program that demonstrates the ? operator. It uses it to obtain the absolute value of a variable.

```
// Demonstrate ?.
class Ternary {
  public static void main(String args[]) {
    int i, k;

    i = 10;
    k = i < 0 ? -i : i; // get absolute value of i
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);

    i = -10;
    k = i < 0 ? -i : i; // get absolute value of i
    System.out.print("Absolute value of ");
    System.out.println(i + " is " + k);
  }
}
```

The output generated by the program is shown here:

```
Absolute value of 10 is 10
Absolute value of -10 is 10
```

➢ Operator Precedence

- Following table describes the precedence of operators. Though parenthesis, square brackets etc. are separators, they do behave like operators in expressions. Operators at same precedence level will be evaluated from left to right, whichever comes first.

Highest			
( )	( )		
++	--	~	!
*	/	%	
+	-		
<<	>>>	>>	
<	>=	>	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			

➢ Using Parentheses

- Parentheses always make the expression within them to execute first. This is necessary sometimes to obtain the result you desire. For example,

  a >> b + 3

- This expression first adds 3 to b and then shifts a right by that result. That is, this expression can be rewritten using redundant parentheses like this:

  a >> (b + 3)

- If you want to first shift a right by b positions and then add 3 to that result, you will need to parenthesize the expression like this: (a >> b) + 3
- Altering the normal precedence of an operator, parentheses can sometimes be used to help clarify the meaning of an expression.
- Parentheses (redundant or not) do not degrade the performance of your program. Therefore, adding parentheses to reduce ambiguity does not negatively affect your program.

➢ Control Statements

- A programming language uses *control* statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: **selection, iteration,** and **jump.**
- *Selection* statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. *Iteration* statements enable program execution to repeat one or more statements (that is, iteration statements form loops). *Jump* statements allow your program to execute in a nonlinear fashion. All of Java's control statements are examined here.

➢ Java's Selection Statements

- Java supports two selection statements: **if** and **switch**. These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

if Statement:

- 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;
```

- Here, if a is less than b, then a is set to zero. Otherwise, b is set to zero. In no case are they both set to zero.
- Most often, the expression used to control the if will involve the relational operators. However, this is not technically necessary. It is possible to control the if using a single boolean variable, as shown in this code fragment:

```
boolean dataAvailable;
// ...
if (dataAvailable)
ProcessData();
else
waitForMoreData()
```

- Both statements within the if block will execute if **bytesAvailable** is greater than zero.

Nested-if Statement:

- A *nested* **if** is an **if** statement that is the target of another **if** or **else**.
- 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)
```

- The final else is not associated with if(j<20) because it is not in the same block (even though it is the nearest if without an else). Rather, the final else is associated with if(i==10). The inner else refers to if(k>100) 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 last else statement is performed. If there is no final else and all other conditions are false, then no action will take place.
- Here is a program that uses an if-else-if ladder to determine which season a particular month is in.

```
// Demonstrate if-else-if statements.
class IfElse {
  public static void main(String args[]) {
    int month = 4; // April
    String season;

    if(month == 12 || month == 1 || month == 2)
      season = "Winter";
    else if(month == 3 || month == 4 || month == 5)
      season = "Spring";
    else if(month == 6 || month == 7 || month == 8)
      season = "Summer";
    else if(month == 9 || month == 10 || month == 11)
      season = "Autumn";
    else
      season = "Bogus Month";

    System.out.println("April is in the " + season + ".");
  }
}
```

Here is the output produced by the program:

```
April is in the Spring.
```

Switch Statement:

- The **switch** statement is Java's multi-way 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.
- The **default** statement is optional. If no **case** matches and no **default** is present, then no further action is taken.
- The break statement is used inside the switch to terminate a statement sequence. When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement. This has the effect of "jumping out" of the switch.

```
// A simple example of the switch.
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.
```

- The **break** statement is optional. If you omit the **break**, execution will continue on into the next **case**. For example, consider the following program:

```
// In a switch, break statements are optional.
class MissingBreak {
  public static void main(String args[]) {
    for(int i=0; i<12; i++)
      switch(i) {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
          System.out.println("i is less than 5");
          break;
        case 5:
        case 6:
        case 7:
        case 8:
        case 9:
          System.out.println("i is less than 10");
          break;
        default:
          System.out.println("i is 10 or more");
      }
  }
}
```

This program generates the following output:

```
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 5
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is less than 10
i is 10 or more
i is 10 or more
```

➢ Iteration Statements

- Java's iteration statements are **for, while, and do-while.** These statements create what we commonly call *loops.* A loop repeatedly executes the same set of instructions until a termination condition is met.

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 the 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. ion 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.

```
// Demonstrate the while loop.
class While {
  public static void main(String args[]) {
    int n = 10;

    while(n > 0) {
      System.out.println("tick " + n);
      n--;
    }
  }
}
```

When you run this program, it will "tick" ten times:

```
tick 10
tick 9
tick 8
tick 7
tick 6
tick 5
tick 4
tick 3
tick 2
tick 1
```

- 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.

do- while Loop:

- The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop. The general form is –

```
do
{
//body of the 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 loop in the preceding program, while technically correct, can be written more efficiently as follows:

```
do {
System.out.println("tick " + n);
} while(--n > 0);
```

- In this example, the expression (--n > 0) combines the decrement of n and the test for zero into one expression. Here is how it works. First, the --n statement executes, decrementing n and returning the new value of n. This value is then compared with zero. If it is greater than zero, the loop continues; otherwise it terminates.

for:

- The general form is –

```
for(initialization; condition; updation)
{
// body of loop
```

}

- 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 *updation* 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.
- Here is a version of the "tick" program that uses a for loop:

```
// Demonstrate the for loop.
class ForTick {
  public static void main(String args[]) {
    int n;

    for(n=10; n>0; n--)
      System.out.println("tick " + n);
  }
}
```

Note: Study the loop variations from the TB.

for-each:

- The for-each style of **for** is also referred to as the *enhanced* **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*. There are various types of collections that can be used with the **for**.
- 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.
- Because the iteration variable receives values from the collection, *type* must be the same as (or

compatible with) the elements stored in the collection. Thus, when iterating over arrays, *type* must be compatible with the base type of the array. Consider an example –

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int i=0; i < 10; i++) sum += nums[i];
```

- The above set of statements can be optimized as follows –

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums)
sum += x;
```

- With each pass through the loop, **x** is automatically given a value equal to the next element in **nums**. Thus, on the first iteration, **x** contains 1; on the second iteration, **x** contains 2; and so on. Not only is the syntax streamlined, but it also prevents boundary errors.

For multi-dimensional arrays:

- The for-each version also works for multi-dimensional arrays. Since a 2-d array is an array of 1-d array, the iteration variable must be a reference to 1-d array. In general, when using the for-each **for** to iterate over an array of *N* dimensions, the objects obtained will be arrays of *N*–1 dimensions. Consider the following example –
- **Demonstration of *for-each* version of *for* loop**

```
class ForEach {
public static void main(String args[]) {
int sum = 0;
int nums[][] = new int[2][3];
// give nums some values
for(int i = 0; i < 2; i++)
for(int j=0; j < 3; j++)
nums[i][j] = (i+1)*(j+1);
for(int x[ ] : nums){          //nums is a 2-d array and x is 1-d array
```

```
for(int y : x) {                    // y refers elements in 1-d array x
System.out.println("Value is: " +y);
sum += y;
} }
System.out.println("Summation: " + sum);
} }
```

The output would be –

```
Value is: 1
Value is: 2
Value is: 3
Value is: 2
Value is: 4
Value is: 6
Summation: 18
```

- The for-each version of for has several applications viz. Finding average of numbers, finding minimum and maximum of a set, checking for duplicate entry in an array, searching for an element in unsorted list etc. The following program illustrates the sequential (linear) search.
- **Linear/Sequential Search Program:**

```
class SeqSearch{
public static void main(String args[]){
int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
int val = 5;
boolean found = false;
for(int x : nums){
if(x == val){
```

```
found = true; break;
}}
if(found)
System.out.println("Value found!");
}}
```

The output would be –

Value found !

➢ Jump Statements

- Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control to another part of your program.

Using break:

- In java, ***break*** can be used in 3 different situations:
  1. To terminate statement sequence in **switch**
  2. To exit from a loop
  3. Can be used as a *civilized* version of ***goto***
- Following is an example showing terminating a loop using break.

```
for (int i=0;i<20;i++)
if(i==5)
break;
else
System.out.println(" i= " + i);
```

- The above code snippet prints values from 0 to 4 and when i become 5, the loop is terminated.

Using break to Exit a Loop

- 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.
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.");
  }
}
```

This program generates the following output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

- Although the for loop is designed to run from 0 to 99, the break statement causes it to terminate early, when i equals 10. 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.");
  }
}
```

- When used inside a set of nested loops, the break statement will only break out of the innermost loop. For example:

```
// Using break with nested loops.
class Breakloop3 {
  public static void main(String args[]) {
    for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
      for(int j=0; j<100; j++) {
        if(j == 10) break; // terminate loop if j is 10
        System.out.print(j + " ");
      }
      System.out.println();
    }
    System.out.println("Loops complete.");
  }
}
```

This program generates the following output:

Pass 0: 0 1 2 3 4 5 6 7 8 9

Pass 1: 0 1 2 3 4 5 6 7 8 9

Pass 2: 0 1 2 3 4 5 6 7 8 9

Loops complete.

- The break statement in the inner loop only causes termination of that loop. The outer loop is unaffected.

Using continue:

- 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. For all three loops, any intermediate code is bypassed.
- For example –

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

- This code uses the % operator to check if i is even. If it is, the loop continues without printing a newline. Here is the output from this program:

0 1

2 3

4 5

6 7

8 9

Using return

- The ***return*** statement is used to explicitly return the method. Based on some condition, we may need to go back to the calling method sometimes. So, we can use ***return*** in such situations.
- 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( ).

```
// Demonstrate return.
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.");
}
}
```

The output from this program is shown here:

```
Before the return.
```

- The final println( ) statement is not executed. As soon as return is executed, control passes back to the caller.