Module 2:
1.History of Java
2. Evolution of java,
3.DataTypes,Variables and Arrays.
4.Operators.
5.Control statements.

Java is an object-oriented programming language developed by Sun Microsystems, a company best known for its high-end Unix workstations.
• Java is modeled after C++
• Java language was designed to be small, simple, and portable across platforms and operating systems, both at the source and at the binary level
• Java also provides for portable programming with applets. Applets appear in a Web page much in the same way as images do, but unlike images, applets are dynamic and interactive.

The C# Connection

- Java's innovative features, constructs, and concepts have become baseline for any new language.
- C# is closely related to Java.Created by Microsoft to support the .NET Framework.
- Both languages share the same general syntax, support distributed programming, and utilize the same object model.
- There are differences between Java and C#, but the overall "look and feel" of these languages is very similar.

How Java Changed the Internet

- Applet changed the way the content can be rendered online.
- Java also addressed issues associated with the Internet: portability and security

Java Applets

- An *applet* is a special kind of Java program that is designed to be transmitted over Internet and automatically executed by a Java-compatible web browser.
- If the user clicks a link that contains an applet, the applet will be automatically downloaded and run in the browser.
- Applets are typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that can execute locally, rather than on the server.
- The applet allows some functionality to be moved from the server to the client.
- In a web page majorly two types of content is rendered.
  - 1st passive information (reading e-mail,is viewing passive data)
  - dynamic, active program(the program's code execution)
- Applet is a dynamic, self-executing program on the client computer, yet it is initiated by the server.

- Dynamic, networked programs are serious problems in the areas of security and portability. As program that downloads and executes automatically on the client computer must be prevented from doing harm.
- It must also be able to run in a variety of different environments and under different operating systems.
- Java solved these problems in an effective and elegant way.

Security

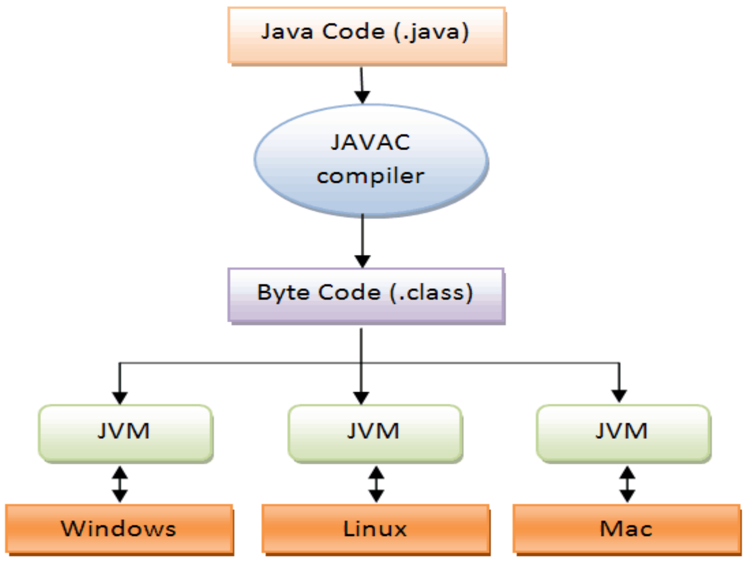
- the code we download might contain virus, Trojan horse, or other harmful code that can gain unauthorized access to system resources.
- For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of computer.
- Java achieved protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer.

Portability

- different types of computers and operating systems connected to internet
- Java program must be able to run on any computer connected to the Internet,
- 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.
- It is not practical to have different versions of the applet for different computers. The *same* code must work on *all* computers.
- Therefore, some means of generating portable executable code was needed. The same mechanism which ensure security also helps in portability.

Java's Magic: The Bytecode

- The key that allows Java to solve both the security and the portability problems is that the output of a Java compiler is not executable code. Rather, it is bytecode.
- *Bytecode* is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the *Java Virtual Machine (JVM).*
- modern programming languages are designed to be compiled into executable code because of performance concerns
- 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.
- 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 fact that a Java program is executed by the JVM also helps to make it secure.
- Because the JVM is in control, it can contain the program and prevent it from generating side effects outside of the system.
- bytecode has been highly optimized, the use of bytecode enables the JVM to execute programs much faster

Servlets: Java on the Server Side

- A servlet is a small program that executes on the server.
- Just as applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server.
- Servlets are used to create dynamically generated content that is then served to the client.
- For example, an online store might use a servlet to look up the price for an item in a database. The price information is then used to dynamically generate a web page that is sent to the browser.
- Servlets increases performance.
- Because servlets (like all Java programs) are compiled into bytecode and executed by the JVM, they are highly portable. Thus, the same servlet can be used in a variety of different server environments.

The Java Buzzwords

Simple

- Java was designed to be easy for the professional programmer to learn and use effectively.
- As Java inherits the C/C++ syntax and many of the object-oriented features of C++, its easy to learn.

Object-Oriented

- The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance nonobjects.

Robust

- the program must execute reliably in a variety of systems. To gain reliability, Java restricts us to find mistakes early in program development.
- As Java is a strictly typed language, it checks code at compile time. also checks code at run time.
- Java programmes behave in a predictable way under diverse conditions is a key feature of Java.
- Programs fail in 2 conditions, memory management mistakes and mishandled exceptional conditio.
- Java virtually eliminates memory management problems by managing memory allocation and deallocation automatically.
- Exceptional conditions(run time errors) in Java is handled well by providing object-oriented exception handling

Multithreaded

- Java programs can do many things simultaneously.
- The Java run-time system supports multiprocess synchronization that enables to construct smoothly running interactive systems.
- Java's easy-to-use approach to multithreading allows to work on specific behavior of the program, not the multitasking subsystem.

Architecture-Neutral

- A central issue for the Java design was that of code longevity and portability.
- As Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. (same program will not execute in different platforms)
- Java Virtual Machine in an attempt to alter this situation. The goal is "write once; run anywhere, any time, forever."

Interpreted and 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.
- 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.

Distributed

- Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols.
- Java supports *Remote Method Invocation (RMI)*. This feature enables a program to invoke methods across a network.

Dynamic

- Java programs carry run-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 manner.
- This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

.

A First Simple Program

```
/* "Example.java". */
class Example
{
        public static void main(String args[])
        {
        System.out.println("This is a simple Java program.");
        }
}
```

.

Compiling the Program in jdk

- the name of the source file should be **Example.java**.
- 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 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**.

  ```
  C:\>java Example
  ```
- When the program is run, the following output is displayed:

```
This is a simple Java program.
```

Explaination

- 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 (}).
- public static void main(String args[]) {
- This line begins the **main( )** method. 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.
- **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. This is necessary since **main( )** is called by the Java Virtual Machine before any objects are made.
- The keyword **void** tells the compiler that **main( )** does not return a value.
- **String args[ ]** declares a parameter named **args**, which is an array of instances of the class **String**.
- **args** receives any command-line arguments present when the program is executed.
- System.out.println("This is a simple Java program.");
- Output is actually accomplished by the built-in **println( )** method, **println( )** displays the string which is passed to it.
- **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console.

2. Variables and Data Types

- Variables are locations in memory in which values can be stored. They have a name, a type, and a value.
- Java has three kinds of variables: instance variables, class variables, and local variables.
- Instance variables, are used to define attributes or the state for a particular object.
- Class variables are similar to instance variables, except their values apply to all that class's instances (and to the class itself) rather than having different values for each object.
- Local variables are declared and used inside method(function) definitions,
- Variable declarations consist of a type and a variable name:
  Examples :int myAge; String myName; boolean value;

The Primitive Types

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

2.1 Integer types.

Type	*Size*	*Range*
byte	8 bits	—128 to 127
short	16 bits	—32,768 to 32,767
int	32 bits	—2,147,483,648 to 2,147,483,647
long	64bits	—9223372036854775808 to 9223372036854775807

// 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;

		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:

```
In 1000 days light will travel about 16070400000000 miles.
```

Clearly, the result could not have been held in an **int** variable.

2.2 Floating-point

- This is used for numbers with a decimal part.
- There are two floating-point types:
  float (32 bits, single-precision) and double (64bits, double-precision).

```
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);
	}
}
```

Output:

```
Area of circle is 366.24
```

2.3 Char

- The char type is used for individual characters. Because Java uses the Unicode character set, the char type has 16 bits of precision, unsigned.

```
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);
          }
}
 output:
ch1 and ch2: X Y

// char variables behave like integers.
class CharDemo2
 {
          public static void main(String args[])
           {
                    char ch1;
                    ch1 = 'X';
                    System.out.println("ch1 contains " + ch1);
                    ch1++; // increment ch1
                    System.out.println("ch1 is now " + ch1);
          }
}
 output:
ch1 contains X
ch1 is now Y
```

2.4 Boolean

- The boolean type can have one of two values, true or false.

```
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);
                    if(b)
                    System.out.println("This is executed.");
                    b = false;
                    if(b)
```

```
                               System.out.println("This is not executed.");
                              System.out.println("10 > 9 is " + (10 > 9));
                    }
    }
     output:
    b is false
    b is true
    This is executed.
    10 > 9 is true
```

2.5 Literals

- Literals are used to indicate simple values in Java programs.
- Number Literals
- There are several integer literals.Ex: 4, is a decimal integer literal of type int
- Floating-point literals usually have two parts: the integer part and the decimal part—Ex: 5.677777.
- Boolean Literals:Boolean literals consist of the keywords true and false.
- These keywords can be used anywhere needed a test or as the only possible values for Boolean variables.

2.6 Character Literals

- Character literals are expressed by a single character surrounded by single quotes: 'a', '#', '3', and so on. Characters are stored as 16-bit Unicode characters.

The Java Class Libraries

- **println( )** and **print( )**. these methods are members of the **System** class, which is a class predefined by Java that is automatically included in your programs.
- 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.

.

Dynamic Initialization of variables.

- 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:

```
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);
        }
}
```

- **sqrt( )**, is a built in method of the **Math** class.

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*. Thus, each time we start a new block, we are creating a new scope.
- A scope determines what objects are visible to other parts of program.
- It also determines the lifetime of those objects.
- In Java, the two major scopes are those defined by a class and those defined by a method.
- In nested scopes 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)
        {
        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);
}
}
```

- 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.

```
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);
                }
        }
}
```

output:

```
        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

- To 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.
- there is no automatic conversion defined from **double** to **byte**.
- It is still possible to obtain a conversion between incompatible types. We 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.
- Ex, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required.
- the numeric types, including integer and floating-point types, are compatible with each other.
- 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

- 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**(narrowing conversion).
- To create a conversion between two incompatible types, we must use a cast.
- A *cast* is simply an explicit type conversion.
- It has this general form:

  (*target-type*) *value*

-
  ```
  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.
- Ex:if the value 1.23 is assigned to an integer, the resulting value will be 1.

```
// 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("\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
```

Automatic Type Promotion in Expressions

- In the following expression:
  ```
  byte a = 40;
  byte b = 50;
  byte c = 100;
  int d = a * b / c;
  ```
- The result of the intermediate term **a * b** easily exceeds the range of either of its **byte** operands. To handle this kind of problem, Java automatically promotes each **byte**, **short**, or **char** operand to **int** when evaluating an expression. This means that the subexpression **a * b** is performed using integers—not bytes.
- For example, this seemingly correct code causes a problem:
  ```
  byte b = 50;
  b = b * 2; // Error! Cannot assign an int to a byte!
  ```
- In such cases we should use an explicit cast, such as
  ```
  byte b = 50;
  b = (byte)(b * 2);
  ```
  which yields the correct value of 100.

The Type Promotion Rules

- First,all **byte**, **short**, and **char** values are promoted to **int**, as just described. Then, if one operand is a **long**, the whole expression is promoted to **long**. If one operand is a **float,** the entire expression is promoted to **float**. If any of the operands is **double**, the result is **double**.
- The following program demonstrates how each value in the expression gets promoted to match the second argument to each binary operator:

```
class Promote
{
        public static void main(String args[])
         {
                byte b = 42;
                char c = 'a';
                short s = 1024;
                int i = 50000;
                float f = 5.67f;
                double d = .1234;
                double result = (f * b) + (i / c) - (d * s);
                System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
                System.out.println("result = " + result);
        }
}
```

- Here,double result = (f * b) + (i / c) - (d * s);
- In the first subexpression, **f * b**, **b** is promoted to a **float** and the result of the subexpression is **float**. Next, in the subexpression **i / c**, **c** is promoted to **int**, and the result is of type **int**.
- Then, in **d * s**, the value of **s** is promoted to **double**, and the type of the subexpression is **double**.
- three intermediate values, **float**, **int**, and **double**, are considered. The outcome of **float** plus an **int** is a **float**. Then the resultant **float** minus the last **double** is promoted to **double**, which is the type for the final result of the expression.

Arrays

- An *array* is a group of like-typed variables that are referred to by a common name.
- Aspecific 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, essentially, a list of like-typed variables.
- The general form of a one-dimensional array declaration is
  type array-var = new *type*[*size*];
- Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array,
- *array-var* is the array variable that is linked to the array.
- The elements in the array allocated by **new** will automatically be initialized to zero.
- using **new** ,allocate the memory that will hold the array
- This example allocates a 12-element array of integers and links them to **month_days**.
  int month_days = new int[12];

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

```
// An improved version of the previous program.
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.");
          }
}
```

Output: April has 30 days.

Example prog that uses a one-dimensional array to find the average of a set of numbers.

```
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;
                    C h a p t e r 3 : D a t a T y p e s , V a r i a b l e s , a n d A r r a y s 51
                    for(i=0; i<5; i++)
                    result = result + nums[i];
                    System.out.println("Average is " + result / 5);
          }
}
```

Multidimensional Arrays

- To declare a multidimensional array variable, specify each additional index using another set of square brackets.
- For example, the following declares a twodimensional array variable called **twoD**.

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

```
// 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
```

- thefollowing code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension manually.

```
// 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
```

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

This program generates the following 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
```

3.Operators

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
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
– =	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

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

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

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

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

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);
        }
}
```

output:

```
x mod 10 = 2
y mod 10 = 2.25
```

Arithmetic Compound Assignment Operators

Operation	Equivalent Operation
a = a + 4;	a += 4;
a = a % 2;	a %= 2;

- the **%=** obtains the remainder of **a**/2 and puts that result back into **a**.

```
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 statement: x = x + 1; can be written as x++;
- The statement x = x - 1; is equivalent to x--;
- These operators are unique where they can appear both in *postfix* form and *prefix* form.
- In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression.
- In postfix form, the previous value is obtained for use in the expression, and then the operand is modified.
- For example:
  ```
  x = 42;
  y = ++x;
  ```
  In this case, **y** is set to 43 because the increment occurs *before* **x** is assigned to **y**.
- Thus, the line **y = ++x;** is the equivalent of these two statements:
  ```
  x = x + 1;
  y = x;
  ```
- Here,
  ```
          x = 42;
          y = x++;
  ```
- the value of **x** is obtained before the increment operator is executed, so the value of **y** is 42.
- Here, the line **y = x++;** is the equivalent of these two statements:
  ```
  y = x;
  x = x + 1;
  ```

```
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**.

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

- Since the bitwise operators manipulate the bits within an integer,
- Ex:, the **byte** value for 42 in binary is 00101010,
- 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.

- For example, –42 is represented as
  00101010
  11010101, then adding 1, which results in
  11010110, or –42.

- a **byte** value, zero is represented by 00000000.
  Inverting, its 11111111 adding 1 results in 100000000.
  where –0 is the same as 0,
- 11111111 is the encoding for –1.

The Bitwise Logical Operators

- The bitwise logical operators are **&**, |, ^, and **~**.
- 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:

```
00101010      (42)
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.
- Ex:

```
  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.

```
    00101010       42
  ^ 00001111       15
    00100101       37
```

Using the Bitwise Logical Operators

The following program demonstrates 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;                    // 0011 in binary
        int b = 6;                     // 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]);
}
}
```

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 operator, **<<**, shifts all of the bits in a value to the left a specified number of times.
- It has this general form: *value* << *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.
- **byte** and **short** values are promoted to **int** when an expression is evaluated. 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**,

```
class ByteShift
 {
        public static void main(String args[])
        {
                byte a = 64, b;
                int i;
                i = a << 2;
                b = (byte) (a << 2);
                System.out.println("Original value of a: " + a);
                System.out.println("i and b: " + i + " " + b);
        }
}
```

The output generated by this program is shown here:

```
Original value of a: 64
i and b: 256 0
```

- Since **a** is promoted to **int** for the purposes of evaluation, left-shifting the value 64 (0100 0000) twice results in **i** containing the value 256 (1 0000 0000). However, the value in **b** contains 0 because after the shift, the low-order byte is now zero. Its only 1 bit has been shifted out.

```
// Left shifting as a quick way to multiply by 2.
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 starting value was carefully chosen so that after being shifted left 4 bit positions, it would produce –32. As you can see, when a 1 bit is shifted into bit 31, the number is interpreted as negative.

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*
- 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*.
- 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 value 35 is shifted 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
```

- When we are shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit. This is called *sign extension* and serves to preserve the sign of negative numbers when we shift them right.
- For example,

```
11111000 –8
>>1
11111100 –4
```

class HexByte

```
{
        Public static void main(String args[])
        {
        byte a=-8;
        byte b = (byte) (a>>1);
        System.out.println("Right shift value is" +b);
        }
}
```

Here is the output of this program:

```
b = -4
```

The Unsigned Right Shift

- the >> operator automatically fills the high-order bit with its previous contents each time a shift occurs. This preserves the sign of the value.
- 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 :

```
11111111 11111111 11111111 11111111          –1
>>>24
00000000 00000000 00000000 11111111          255
```

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.

  ```
  a = a >> 4;
  a >>= 4;
  ```

- Likewise, the following two statements are equivalent:

  ```
  a = a | b;
  a |= b;
  ```

```
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.

Operator	Result
==	Equal to
!=	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.
- only integer, floating-point, and character operands may be compared to see which is greater or less than the other.
- 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**.

Boolean Logical Operators

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

Operator	Result
&	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

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

```
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);
	}
}
```

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

Short-Circuit Logical Operators

- There are secondary versions of the Boolean AND and OR operators, and are known as *short-circuit* logical operators.
- 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.(therefore there is no need to evaluate the second operand.)
- Short circuit logical operators are the || and **&&** f

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.*
- 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.

.

The ? Operator

- Java provides *ternary* (three-way) *operator* that can replace certain types of if-then-else statements.
- 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.

.

```
class Ternary
{
        public static void main(String args[])
        {
                int a=5,b=10;
                int c= a>b? a: b;
                System.out.println("bigger number is " +c);
        }
}
 output :
        bigger number is 10.
```

Operator Precedence

Highest			
( )	[ ] .		
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			

?:
= op=
Lowest

Control Statements

- A programming language uses *control* statements to cause the flow of execution to advance and branch into different part of a program.
- Java's program control statements can be put into the following categories:
  - selection,
  - iteration, and
  - jump.
- *Selection* statements allow 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.
- *Jump* statements allow program to execute in a nonlinear fashion.

Java's Selection Statements

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

if

- **if** statement is Java's conditional branch statement.
- General form of **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.
- 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, :

```
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.**
- 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)      // this if is
                 c = d;
                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(j<20)** because it is not in the same block (even though it is the nearest **if** without an **else**)
- . 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 **if**s is the *if-else-if ladder*.
- General form:

```
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.

```
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 + ".");
          }
}
```

output:

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

switch

- The **switch** statement is Java's multiway branch statement.
- It provides an easy way to dispatch execution to different parts of code based on the value of an expression.
- It 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.
- Each **case** value must be a unique literal (that is, it must be a constant, not a variable).
- Duplicate **case** values are not allowed.
- 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.

- The **break** statement is used inside the **switch** to terminate a statement sequence.

```
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.");
                    }
          }
}
```

output:

```
                              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 we omit the **break**, execution will continue on into the next **case**. It is sometimes desirable to have multiple **case**s without **break** statements between them.

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

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
```

```
class Switch
{
        public static void main(String args[])
        {
                int month = 4;
                String season;
                switch (month)
                 {
                case 12:
                case 1:
                case 2:
                season = "Winter";
                break;
                case 3:
                case 4:
                case 5:
                season = "Spring";
                break;
                case 6:
                case 7:
                case 8:
                season = "Summer";
                break;
                case 9:
                case 10:
                case 11:
                season = "Autumn";
                break;
                default:
                season = "Bogus Month";
                }
                System.out.println("April is in the " + season + ".");
        }
}
```

Nested switch Statements

- **switch can be used** as part 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**.

```
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.
• 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 **if**s.

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 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.

```
class While
{
        public static void main(String args[])
         {
                int n = 5;
                while(n > 0)
                 {
                System.out.println("tick " + n);
                n--;
                }
        }
}
```

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

```
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.
- 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");
```

- The body of the **while** (or any other of Java's loops) can be empty. This is because a *null Statement* is syntactically valid in Java.
- For example,

```
class NoBody
{
        public static void main(String args[])
         {
                int i, j;
                i = 100;
                j = 200;
                // find midpoint between i and j
                while(++i < --j) ; // no body in this loop
                System.out.println("Midpoint is " + i);
        }
}
```

This program finds the midpoint between **i** and **j**.
output: Midpoint is 150

do-while

- if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all.
- sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with.
- In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning.
- 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.

```
class DoWhile
{
        public static void main(String args[])
        {
                int n = 5;
                do {
                System.out.println("tick " + n);
                n--;
                } while(n > 0);
        }
}

class Menu
{
        public static void main(String args[]) throws java.io.IOException
        {
                char choice;
                        do
                         {
                        System.out.println("Help on:");
                        System.out.println(" 1. good");
                        System.out.println(" 2. better");
                        System.out.println(" 3. best");
                        System.out.println("4. Excellent");
                        System.out.println("Choose one:");
                        choice = (char) System.in.read();
                        } while( choice < '1' || choice > '4');
                System.out.println("\n");
                switch(choice)
                 {
                        case '1':       System.out.println("Good");
                                        break;
                        case '2':       System.out.println("better");
                                        break;
                        case '3':       System.out.println("best");
                                        break;
                        case '4':       System.out.println("Excellent");
                                        break;
                }
        }
}
```

Here is a sample run produced by this program:

```
Help on:
1. good
2. better
3. best
4. Excellent

Choose one:
4
Excellent.
```

for

- there are two forms of the **for** loop.
- 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.
- general form of the traditional **for** statement:

```
for(initialization; condition; iteration) {
// body
}
```

```
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

- Often the variable that controls a **for** loop is only needed for the purposes of the loop and is not used elsewhere.
- When this is the case, it is possible to declare the variable inside the initialization portion of the **for**.
- For example,the loop control variable **n** is declared as an **int** inside the **for**:

```
class ForTick
 {
	public static void main(String args[])
	 {
		for(int n=10; n>0; n--)
		System.out.println("tick " + n);
	}
}
```

```
class FindPrime
{
	public static void main(String args[])
	 {
		int num;
		boolean isPrime = true;
		num = 14;
		for(int i=2; i <= num/i; i++)
		 {
			if((num % i) == 0)
			{
			isPrime = false;
```

```
                        break;
                        }
                }
                if(isPrime)
                 System.out.println("Prime");
                else
                System.out.println("Not Prime");
        }
}
```

Using the Comma

- There will be times when you will want to include more than one statement in the initialization and iteration portions of the **for** loop.
- For example, consider the loop in the following program:

```
class Comma
{
        public static void main(String args[])
         {
                int a, b;
                for(a=1, b=4; a<b; a++, b--)
                 {
                        System.out.println("a = " + a);
                        System.out.println("b = " + b);
                }
        }
}
```

- the initialization portion sets the values of both **a** and **b**. The two comma separated statements in the iteration portion are executed each time the loop repeats.
- output:

```
a = 1
b = 4
a = 2
b = 3
```

Some for Loop Variations

- The **for** loop supports a number of variations that increase its power and applicability.

```
class ForVar
 {
        public static void main(String args[])
         {
                int i;
                boolean done = false;
                i = 0;
                for( ; !done; )
                 {
                        System.out.println("i is " + i);
                        if(i == 10) done = true;
                        i++;
                  }
            }
      }
```

- Here, the initialization and iteration expressions have been moved out of the **for**. Thus, parts of the **for** are empty.

The For-Each Version of the for Loop

- The advantage of this approach is that no new keyword is required, and no preexisting code is broken.
- The for-each style of **for** is also referred to as the *enhanced* **for** loop.
- The general form for-each version of the **for** is shown here:
  for(*type itr-var : collection*) *statement-block*
- Here, *type* specifies the type
- *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**, but the only type used here is the array..

```
class ForEach
{
      public static void main(String args[])
       {
            int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            int sum = 0;

            for(int x : nums)
            {
                  System.out.println("Value is: " + x);
                  sum += x;
            }
      System.out.println("Summation: " + sum);
      }
}
```

The output from the program is shown here.

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55
```

- the for-each **for** loop iterates until all elements in an array have been examined, it is possible to terminate the loop early by using a **break** statement.

```
class ForEach2
 {
        public static void main(String args[])
        {
                int sum = 0;
                int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

                for(int x : nums)
                 {
                        System.out.println("Value is: " + x);
                        sum += x;
                        if(x == 5) break; // stop the loop when 5 is obtained
                }
                System.out.println("Summation of first 5 elements: " + sum);
                }
}
```

output :

```
                Value is: 1
                Value is: 2
                Value is: 3
                Value is: 4
                Value is: 5
                Summation of first 5 elements: 15
```

Nested Loops

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

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

The output produced by this program is shown here:

```
..........
.........
........
.......
......
.....
....
...
..
.
```

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, the **break** statement has three uses.
- First, as you have seen, it terminates a statement sequence in a **switch** statement.
- Second, it can be used to exit a loop.
- Third, it can be used as a “civilized” form of goto.

Using break to Exit a Loop

- By using **break**, we 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.

```
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.
```

- We can Use break to exit a while loop.
- When used inside a set of nested loops, the **break** statement will only break out of the innermost loop.

```
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.
```

Using break as a Form of Goto

- 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.
- 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. When this form of **break** executes, control is transferred out of the named block.

```
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.");
		}
	}
}
```

output:

```
Before the break.
This is after second block.
```

- One of the most common uses for a labeled **break** statement is to exit from nested loops.

```
class BreakLoop4
 {
	public static void main(String args[])
	 {
		outer: for(int i=0; i<3; i++)
		{
			System.out.print("Pass " + i + ": ");
			for(int j=0; j<100; j++)
			 {
				if(j == 10) break outer; // exit both loops
				System.out.print(j + " ");
			}
		System.out.println("This will not print");
```

```
                }
          System.out.println("Loops complete.");
          }
}
```

This program generates the following output:
Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete.

Using continue

- Here the loop will skip the execution of a particular iteration upen certain condition and continue to execute further iteration.

```
class Continue
{
          public static void main(String args[])
           {
                    for(int i=0; i<10; i++)
                     {
                              if (i == 2) continue;
                              System.out.print(i + " ");
                    }
          }
}
 output :
                    0 1 3 4 5 6 7 8 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();
          }
}
```

The **continue** statement in this example terminates the loop counting **j** and continues with the next iteration of the loop counting **i**. Here is the output of this program:

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

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.
- 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.");
	}
}
```

output:

```
		Before the return.
```

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