UNIT I

Introduction to Java: The key attributes of object oriented programming, Simple program, The Java keywords, Identifiers, Data types and operators, Program control statements, Arrays, Strings, String Handling

Introduction:

- JAVA is a programming language.
- Computer language innovation and development occurs for two fundamental reasons:
  - To adapt to changing environments and uses
  - To implement refinements and improvements in the art of programming
- Java is related to C++, which is a direct descendant of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java's object-oriented features were influenced by C++.
- Definition: Object-oriented programming (OOP) is a programming methodology that helps organize complex programs through the use of inheritance, encapsulation, and polymorphism.
- Java was developed by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. This language was initially called "Oak," but was renamed "Java" in 1995.
- Java was not designed to replace C++. Java was designed to solve a certain set of problems. C++ was designed to solve a different set of problems.
- Java contribution to internet:
  Java programming had a profound effect on internet.
- **Java Applets**
  An *applet* is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser.
- **Security**
  Every time you download a "normal" program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code.
  In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack.
  Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer.

- **Portability**

  Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. Java programming provide portability

  Byte code:

  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)*. In essence, the original JVM was designed as an *interpreter for bytecode.*

- **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**.**

- **The Java Buzzwords**
  - Simple
  - Secure
  - Portable
  - Object-oriented
  - Robust
  - Multithreaded
  - Architecture-neutral
  - Interpreted
  - High performance
  - Distributed
  - Dynamic

- **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.

  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.

  Some programs are written around “what is happening” and others are written around “who is being affected.” These are the two paradigms that govern how a program is constructed.

  The first way is called the *process-oriented model*. The process-oriented model can be thought of as *code acting on data*. Procedural languages such as C employ this model to considerable success.

  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.*

The key Attributes of OOP:

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

Encapsulation

- ✓ *Encapsulation* is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.
- ✓ In Java, the basis of encapsulation is the class.
- ✓ A *class* defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class. Objects are sometimes referred to as *instances of a class.*
- ✓ Thus, a class is a logical construct; an object has physical reality.
- ✓ The code and data that constitute a class are called *members* of the class. Specifically, the data defined by the class are referred to as *member variables* or *instance variables.* The code that operates on that data is referred to as *member methods* or just *methods*
- ✓ 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

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.
- ✓ Inheritance interacts with encapsulation as well. If a given class encapsulates some attributes, then any subclass will have the same attributes *plus* any that it adds as part of its specialization
- ✓ A new subclass inherits all of the attributes of all of its ancestors.

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 {
  // A Java program begins with a call to main( ).
  public static void main(String[ ] args) {
    System.out.println("Java drives the Web.");
  }
}
```

Entering the program:

The first step in creating a program is to enter its source code into the computer.

The name you give to a source file is very important. In Java, a source file is officially called a *compilation unit*. It is a text file that contains (among other things) one or more class definitions. The Java compiler requires that a source file use the **.java** filename extension

The name of the main class should match the name of the file that holds the program.

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:

```
|  javac Example.java
```

The **javac** compiler creates a file called **Example.class** that contains the bytecode version of the program. The output of **javac** is not code that can be directly executed.

Running the program

To actually run the program, you must use the Java application launcher called **java.**

To do so, pass the class name Example as a command-line argument, as shown here:

```
| java Example
```

When the program is run, the following output is displayed:

```
| Java drives the Web.
```

First simple program line by line

The program begins with the following lines:

```
/*

This is a simple Java program.

Call this file "Example.java".

*/
```

This is a *comment*. The contents of a comment are ignored by the compiler. This is multiline comment

```
class Example {
```

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

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

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

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

This line begins the **main( )** method. All Java applications begin execution by calling **main( )**.

The **public** keyword is an *access modifier*, 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** simply tells the compiler that **main( )** does not return a value.

In **main( )**, there is only one parameter, **String args[ ]** declares a parameter named **args**, which is an array of instances of the class **String**. 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("Java drives the Web.");
```

This line outputs the string “Java drives the Web.” followed by a new line on the screen. Output is actually accomplished by the built-in **println( )** method. In this case, **println( )** displays the string which is passed to it. The line begins with **System.out**. **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console.

Example2:

```
/*
   This demonstrates a variable.
   Call this file Example2.java.
*/
class Example2 {
  public static void main(String[ ] args) {
    int var1; // this declares a variable
    int var2; // this declares another variable
    var1 = 1024; // this assigns 1024 to var1
    System.out.println("var1 contains " + var1);
    var2 = var1 / 2;
    System.out.print("var2 contains var1 / 2: ");
    System.out.println(var2);
  }
}
```

O/P:

```
var1 contains 1024
var2 contains var1 / 2: 512
```

Example3:

```
/*
   This program illustrates the differences between int and double.
   Call this file Example3.java.
*/
class Example3 {
  public static void main(String[ ] args) {
    int w; // this declares an int variable
    double x; // this declares a floating-point variable
    w = 10; // assign w the value 10

    x = 10.0; // assign x the value 10.0
    System.out.println("Original value of w: " + w);
    System.out.println("Original value of x: " + x);
    System.out.println(); // print a blank line
    // now, divide both by 4
    w = w / 4;
    x = x / 4;
    System.out.println("w after division: " + w);
    System.out.println("x after division: " + x);
  }
}
```

O/P

```
Original value of w: 10
Original value of x: 10.0

w after division: 2
x after division: 2.5
```

Example:

```
/*
   Try This 1-1
   This program converts gallons to liters.
   Call this program GalToLit.java.
*/
class GalToLit {
  public static void main(String[ ] args) {
    double gallons; // holds the number of gallons
    double liters; // holds conversion to liters

    gallons = 10; // start with 10 gallons
    liters = gallons * 3.7854; // convert to liters
    System.out.println(gallons + " gallons is " + liters + " liters.");
  }
}
```

O/P:

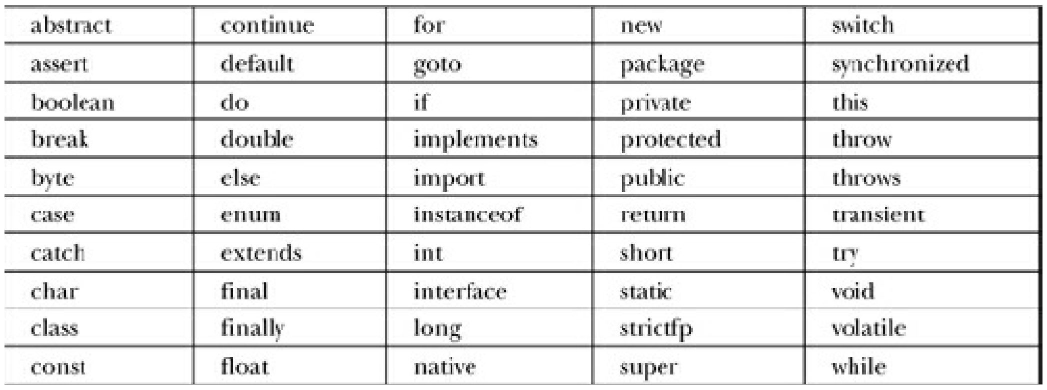
```
10.0 gallons is 37.854 liters.
```

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 identifiers. Thus, they 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

Identifiers

Identifiers are used to name things, such as classes, variables, and methods. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. (The dollar-sign character is not intended for general use.) They must not begin with a number. Java is case-sensitive, so **VALUE** is a different identifier than **Value**. Some examples of valid identifiers are GalToLit, Test, x, y2, maxLoad, my_var.

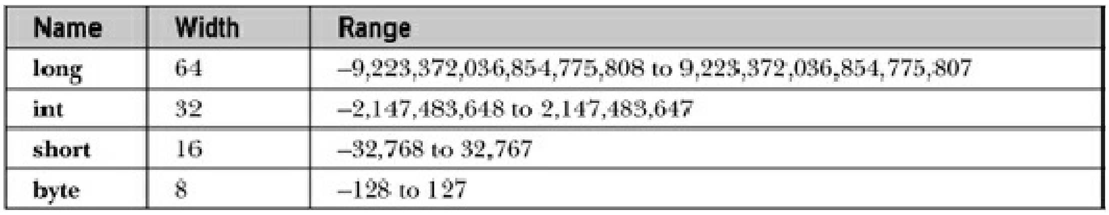
Invalid identifier names include these: 12x, not/ok.

The 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.

All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers.



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

```
// Compute distance light travels using long variables.
import java.util.Scanner;
class Light {
public static void main(String args[ ]) {
        int days;
        long lightspeed;
        long seconds;
        long distance;
        // approximate speed of light in miles per second
        lightspeed = 186000;
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter number of days");
        days=sc.nextInt( );
        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.");
  }
}
```

O/P:

```
Enter number of days
10
In 10 days light will travel about 160704000000 miles.
```

• **Floating-point numbers** This group includes **float** and **double**, which represent numbers with fractional precision.

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

```
// Compute the area of a circle.
import java.util.Scanner;
class Area {
        public static void main(String args[]) {
        double pi, r, a;
        Scanner input= new Scanner(System.in);
        pi = 3.1416; // pi, approximately
        System.out.println("Enter radius ");
        r=input.nextDouble();
        a = pi * r * r; // compute area
        System.out.println("Area of circle is " + a);
  }
}
```

O/P:

```
Enter radius
10.8
Area of circle is 366.436224
```

• **Characters** This group includes **char**, which represents symbols in a character set, like letters and numbers.

In Java **char** is a 16-bit type. The range of a **char** is 0 to 65,535. There are no negative **char**s.

```
// Character variables can be handled like integers.
class CharArithDemo {
  public static void main(String[] args) {
    char ch;
    ch = 'X';
    System.out.println("ch contains " + ch);
    ch++; // increment ch
    System.out.println("ch is now " + ch);
    ch = 90; // give ch the value Z
    System.out.println("ch is now " + ch);
  }
}
```

O/P:

```
ch contains X
ch is now Y
ch is now Z
```

• **Boolean** This group includes **boolean**, which is a special type for representing true/false values.

```
// Demonstrate boolean values.
class BoolDemo {
  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));
  }
}
```

O/P:

```
b is false
b is true
This is executed.
10 > 9 is true
```

Literals are also commonly called constants

Java provides special escape sequences sometimes referred to as backslash character constants

Escape Sequence	**Character**
\n	newline
\t	tab
\b	backspace
\f	form feed
\r	return
\"	" (double quote)
\'	' (single quote)
\\\\	\\ (back slash)
\uDDDD	character from the Unicode character set (DDDD is four hex digits)

Operators:

An operator is a symbol that tells the compiler to perform a specific mathematical, logical, or other manipulation. Java has four general classes of operators: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations.

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(also unary plus)
-	Subtraction(also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
--	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**.

When the division operator is applied to an integer type, there will be no fractional component attached to the result. The modulus operator, **%**, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types.

```
// Demonstrate the % operator.
class ModDemo {
  public static void main(String[ ] args) {
    int iresult, irem;
    double dresult, drem;
    iresult = 10 / 3;
    irem = 10 % 3;
    dresult = 10.0 / 3.0;
    drem = 10.0 % 3.0;
    System.out.println("Result and remainder of 10 / 3: " + iresult + " " + irem);
    System.out.println("Result and remainder of 10.0 / 3.0: " + dresult + " " + drem);
  }
}
```

O/P:

```
Result and remainder of 10 / 3: 3 1
Result and remainder of 10.0 / 3.0: 3.3333333333333335 1.0
```

Increment and Decrement

The increment operator increases its operand by one. The decrement operator decreases its operand by one. For example, this statement: x = x + 1;

can be rewritten like this by use of the increment operator: x++;

Similarly, this statement: x = x - 1;

is equivalent to x--;

Both increment and decrement operators can either prefix or postfix the operand.

There is no difference between the prefix and postfix forms. However, when the increment and/or decrement operators are part of a larger expression, there is an important difference between these two forms appears. 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

EG:

x=10;

y=++x;

in this case y will be set to 11

x=10;

y=x++;

then y will be set to 10

```
// Demonstrate ++.
class IncDec {
	public static void main(String args[ ]) {
		int a = 1;
		int b = 2;
		int c;
		int d;
		c = ++b;
		d = a++;
		c++;
		System.out.println("a = " + a);
		System.out.println("b = " + b);
		System.out.println("c = " + c);
		System.out.println("d = " + d);
	}
}
```

The output of this program follows:

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

Relational and Logical Operators

The *relational operators* determine the relationship that one operand has to the other.

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

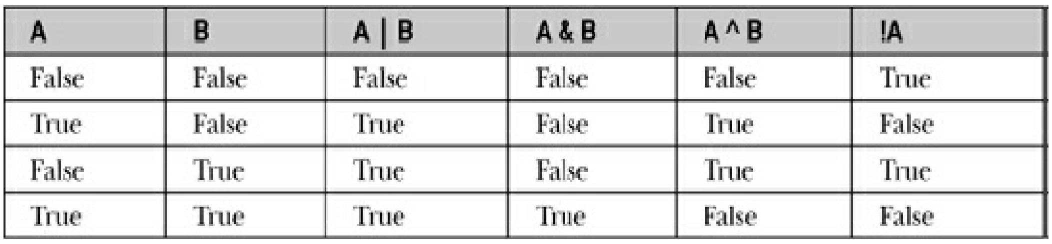
The outcome of these operations is a **boolean** value.

Java does not define true and false in the same way as C/C++. In C/ C++, true is any nonzero value and false is zero. In Java, **true** and **false** are nonnumeric values that do not relate to zero or nonzero. Therefore, to test for zero or nonzero, you must explicitly employ one or more of the relational operators

Logical operators combine two **boolean** values to form a resultant **Boolean** value.

Operator	Meaning
&	AND
|	OR
^	XOR
||	Short-circuit OR
&&	Short-circuit AND
!	NOT

The logical Boolean operators, **&**, |, and ^, operate on **boolean** values in the same way that they operate on the bits of an integer. The logical **!** operator inverts the Boolean state: **!true == false** and **!false == true**. The following table shows the effect of each logical operation:



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

```
// Demonstrate the relational and logical operators.
class RelLogOps {
  public static void main(String [ ] args) {
    int i, j;
    boolean b1, b2;
    i = 10;
    j = 11;
    if(i < j)
            System.out.println("i < j");
    if(i <= j)
            System.out.println("i <= j");
    if(i != j)
            System.out.println("i != j");
    if(i == j)
            System.out.println("this won't execute");
    if(i >= j)
            System.out.println("this won't execute");
    if(i > j)
            System.out.println("this won't execute");
    b1 = true;
    b2 = false;
    if(b1 & b2)
            System.out.println("this won't execute");
    if(!(b1 & b2))
            System.out.println("!(b1 & b2) is true");
    if(b1 | b2)
             System.out.println("b1 | b2 is true");
    if(b1 ^ b2)
            System.out.println("b1 ^ b2 is true");
  }
}
```

O/P:

```
i < j
i <= j
i != j
!(b1 & b2) is true
b1 | b2 is true
b1 ^ b2 is true
```

Short-circuit Logical operators:

These are secondary versions of the Boolean AND and OR operators, and are commonly known as *short-circuit* logical operators.

The difference between normal and short-circuit versions is that the normal operands will always evaluate each operand, but short-circuit versions will evaluate the second operand only when necessary.

When the right-hand operand depends on the value of the left one in order to function properly. For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

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

Since the short-circuit form of AND (**&&**) is used, there is no risk of causing a run-time exception when **denom** is zero. If this line of code were written using the single **&**version of AND, both sides would be evaluated, causing a run-time exception when **denom** is zero.

```
// Demonstrate the short-circuit operators.
class SCops {
  public static void main(String[] args) {
    int n, d, q;
    n = 10;
    d = 2;
    if(d != 0 && (n % d) == 0)
      System.out.println(d + " is a factor of " + n);
    d = 0; // now, set d to zero
    // Since d is zero, the second operand is not evaluated.
    if(d != 0 && (n % d) == 0)
      System.out.println(d + " is a factor of " + n);
   /* Now, try same thing without short-circuit operator.
       This will cause a divide-by-zero error.
    */
    if(d != 0 & (n % d) == 0)
      System.out.println(d + " is a factor of " + n);
  }
}
```

Assignment operators:

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.

Java provides special operators that can be used to combine an arithmetic operation with an assignment.

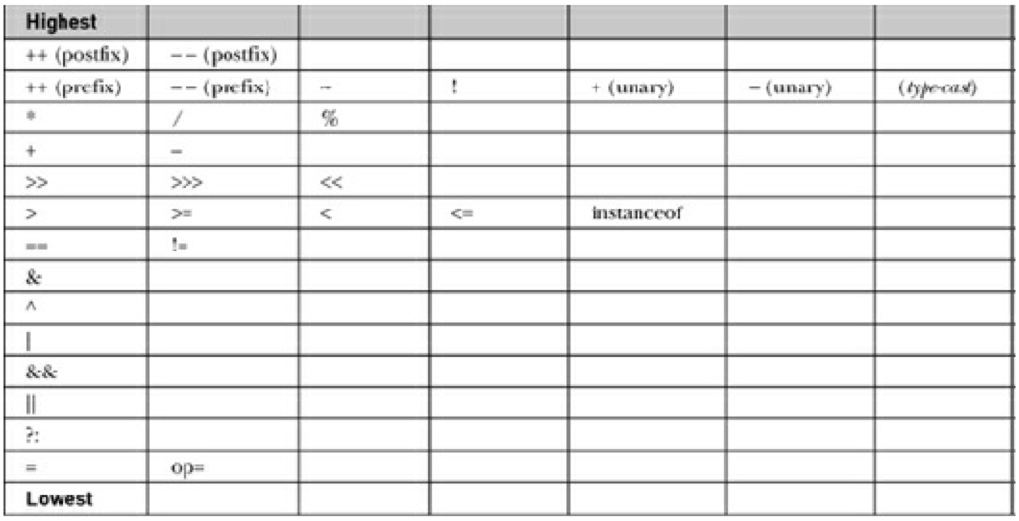
a = a + 4;   can rewrite as: a += 4;

There are compound assignment operators for all of the arithmetic, binary operators.

Thus, any statement of the form var = *var op expression*; can be rewritten as *var op= expression*;

Operator Precedence

The following table shows the order of precedence for Java operators, from highest to lowest. Operators in the same row are equal in precedence. In binary operations, the order of evaluation is left to right (except for assignment, which evaluates right to left). Although they are technically separators, the **[ ]**, **( )**, and **.** can also act like operators. In that capacity, they would have the highest precedence.



Highest						
++ (postfix)	-- (postfix)					
++ (prefix)	-- (prefix)	~	!	+ (unary)	– (unary)	(*type-cast*)
*	/	%				
+	–					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
|						
&&						
||						
?:						
=	op=					
Lowest						

The Precedence of the Java Operators

Using Parentheses

Parentheses raise the precedence of the operations that are inside them.

The Bitwise Operators

Java defines several *bitwise operators* that can be applied to the integer types, **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands.

They are summarized in the following table:

Operator	Result
~	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

The Bitwise Logical Operators

The bitwise logical operators are **&**, |, ^, and **~**. The following table shows the outcome of each operation.

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

```
// Uppercase letters.
class UpCase {
  public static void main(String[] args) {
    char ch;
    for(int i=0; i < 10; i++) {
      ch = (char) ('a' + i);
      System.out.print(ch);
      // This statement turns off the 6th bit.
      ch = (char) ((int) ch & 65503); // ch is now uppercase
      System.out.print(ch + " ");
    }
  }
}
```

O/P:

```
aA  bB  cC  dD  eE  fF  gG  hH  iI  jJ
```

```
// Lowercase letters.
class LowCase {
  public static void main(String[] args) {
    char ch;
    for(int i=0; i < 10; i++) {
      ch = (char) ('A' + i);
      System.out.print(ch);
      // This statement turns on the 6th bit.
      ch = (char) ((int) ch | 32); // ch is now lowercase
      System.out.print(ch + " ");
    }
  }
}
O/P:
Aa  Bb  Cc  Dd  Ee  Ff  Gg  Hh  Ii  Jj
```

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.*

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*.

The ? Operator

Java includes a special *ternary* (three-way) *operator* that can replace certain types of if-then- else statements. This operator is the **?**.

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.

```
import java.util.Scanner;
public class Largest
{
    public static void main(String[] args)
    {
        int a, b, c, d;
        Scanner s = new Scanner(System.in);
        System.out.println("Enter all three numbers:");
        a = s.nextInt( );
        b = s.nextInt( );
        c = s.nextInt( );
        d = a>b?(a>c?a:c):(b>c?b:c);
        System.out.println("Largest of "+a+","+b+","+c+" is: "+d);
    }
}
```

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.

Input characters from the keyboard

To read a character from keyboard we can use **System.in.read ( ).** The **read( )** waits until the user presses a key and then returns the result. The character returned as an integer, so it must be cast into a **char** to assign it to a char variable.

```
// Read a character from the keyboard.
class KbIn {
  public static void main(String[ ] args)
    throws java.io.IOException {
    char ch;
    System.out.print("Press a key followed by ENTER: ");
    ch = (char) System.in.read( ); // get a char
    System.out.println("Your key is: " + ch);
  }
}
```

O/P:

```
Press a key followed by ENTER: k
Your key is: k
```

The above program uses throws java.io.IOException .This line is necessary to handle input errors. It is a part of java exception handling mechanism.

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.

```
// Guess the letter game.
class Guess {
  public static void main(String[ ] args)
    throws java.io.IOException {
    char ch, answer = 'K';
    System.out.println("I'm thinking of a letter between A and Z.");
    System.out.print("Can you guess it: ");
    ch = (char) System.in.read( ); // read a char from the keyboard
    if(ch == answer) System.out.println("** Right **");
  }
}
```

The next version uses **else** to print a message when the wrong letter is picked

```
// Guess the letter game, 2nd version.
class Guess2 {
  public static void main(String[ ] args)
    throws java.io.IOException {
    char ch, answer = 'K';
    System.out.println("I'm thinking of a letter between A and Z.");
    System.out.print("Can you guess it: ");
    ch = (char) System.in.read( ); // get a char
    if(ch == answer) System.out.println("** Right **");
    else System.out.println("...Sorry, you're wrong.");
  }
}
```

Nested ifs

A nested **if** is an **if** statement that is the target of another if or else. When you nest **if**s, 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**.

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. 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;

```
// Demonstrate an if-else-if ladder.
class Ladder {
  public static void main(String[] args) {
    int x;
    for(x=0; x<6; x++) {
      if(x==1)
        System.out.println("x is one");
      else if(x==2)
        System.out.println("x is two");
      else if(x==3)
        System.out.println("x is three");
      else if(x==4)
        System.out.println("x is four");
      else
        System.out.println("x is not between 1 and 4");
    }
  }
}
```

O/P:

```
x is not between 1 and 4
x is one
x is two
x is three
x is four
x is not between 1 and 4
```

switch

The switch provides for a multi-way branch. 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
}
```

```
// Demonstrate the switch.
class SwitchDemo {
  public static void main(String[ ] args) {
    int i;
    for(i=0; i<10; 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;
        case 4:     System.out.println("i is four");
                    break;
        default:     System.out.println("i is five or more");
      }
  }
}
```

The **break** statement is optional. If you omit the **break**, execution will continue on into the next **case**.

Nested switch Statements

We can use a **switch** as part of the statement sequence of an outer **switch**. This is called a *nested* **switch**. Since a **switch** statement defines its own block, no conflicts arise between the **case** constants in the inner **switch** and those in the outer **switch**. For example, the following fragment is perfectly valid:

```
switch(count) {
case 1:switch(target) { // nested switch
            case 0: System.out.println("target is zero");
                    break;
            case 1: // no conflicts with outer switch
                    System.out.println("target is one");
                    break;
            }
            break;
case 2: // …
```

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

• The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of Boolean expression. That is, the **switch** looks only for a match between the value of the expression and one of its **case** constants.

• No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement and an enclosing outer **switch** can have **case** constants in common.

• A **switch** statement is usually more efficient than a set of nested **if**s.

Iteration Statements

Java's iteration statements are **for**, **while**, and **do-while.**

while

The **while** loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
while(condition) {
// body of loop
}
```

The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated

```
// Demonstrate the while loop.
class WhileDemo {
  public static void main(String[ ] args) {
    char ch;
    // print the alphabet using a while loop
    ch = 'a';
    while(ch <= 'z') {
      System.out.print(ch);
      ch++;
    }
  }
}
```

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);
```

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

for loop

The general form of the traditional for statement:

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

```
// Show square roots of 1 to 9.
class SqrRoot {
  public static void main(String[ ] args) {
    double num, sroot;
    for(num = 1.0; num < 10.0; num++) {
      sroot = Math.sqrt(num);
      System.out.println("Square root of " + num +
                          " is " + sroot);
    }
  }
}
```

Some variations on for loop:

- It is possible to declare the variable inside the initialization portion of the **for**.

```
// compute the sum and product of the numbers 1 through 5
for(int i = 1; i <= 5; i++) {
  sum += i; // i is known throughout the loop
  product *= i;
}
```

When you declare a variable inside a **for** loop, there is one important point to remember: the scope of that variable ends when the **for** statement does.

- When using multiple loop control variables the initialization and iteration expressions for each variable are separated by commas.

```
for(i=0, j=10; i < j; i++, j--)
  System.out.println("i and j: " + i + " " + j);
```

- It is possible for any or all of the initialization, condition, or iteration portions of the **for** loop to be blank.

```
  i = 0; // move initialization out of loop
for(; i < 10; ) {
  System.out.println("Pass #" + i);
  i++; // increment loop control var
}
```

NESTED LOOPS:

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

```
for(i=0; i<=5; i++) {
        for(j=1; j<=i; j++)
                System.out.print("*");
        System.out.println( );
}
```

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**, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.

```
// Using break to exit a loop.
class BreakDemo {
  public static void main(String[ ] args) {
    int num;
    num = 100;
    // loop while i-squared is less than num
    for(int i=0; i < num; i++) {
      if(i*i >= num) break; // terminate loop if i*i >= 100
      System.out.print(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

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.

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

break *label*;

// Using break with a label.

```
class Break4 {
  public static void main(String[ ] args) {
    int i;
    for(i=1; i<4; i++) {
one:  {
two:    {
three:    {
            System.out.println("\ni is " + i);
            if(i==1) break one;
            if(i==2) break two;
            if(i==3) break three;
            // this is never reached
            System.out.println("won't print");
          }
          System.out.println("After block three.");
        }
        System.out.println("After block two.");
      }
      System.out.println("After block one.");
    }
    System.out.println("After for.");
  }
}
```

Using continue

It is possible to force an early iteration of a loop, bypassing the loop's normal control structure. This is accomplished using **continue.** The continue statement forces the next iteration of the loop to take place, skipping any code between itself and the conditional expression that controls the loop.

```
// Use continue.
class ContDemo {
  public static void main(String[ ] args) {
    int i;
    // print even numbers between 0 and 100
    for(i = 0; i<=100; i++) {
      if((i%2) != 0) continue; // iterate
      System.out.println(i);
    }
  }
}
```

As with the **break** statement, **continue** may specify a label to describe which enclosing loop to continue. Here is an example program that uses **continue** to print a triangular multiplication table for 0 through 9:

```
// Using continue with a label.
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:

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.

ARRAYS:

An *array* is a collection of variables of same type, referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

One-Dimensional Arrays

A *one-dimensional array* is, essentially, a list of like-typed variables.

The general form to declare a one-dimensional array:

type[ ] array-name=new type[size];

Since arrays are implemented as objects, the creation of an array is a two-step process. First declare an array reference variable. Second allocate memory for the array, assigning the reference to that memory to the array. Thus arrays in java are dynamically allocated using **new** operator.

Eg : int[ ] sample=new int[10];

It is possible to break the above declaration.

int[ ] sample;

sample=new int[10];

// Demonstrate a one-dimensional array.

```
class ArrayDemo {
  public static void main(String[ ] args) {
    int[ ] sample = new int[10];
    int i;
    for(i = 0; i < 10; i = i+1)
      sample[i] = i;
    for(i = 0; i < 10; i = i+1)
      System.out.println("This is sample[" + i + "]: " + sample[i]);
  }
}
```

O/P:

```
This is sample[0]: 0
This is sample[1]: 1
This is sample[2]: 2
This is sample[3]: 3
This is sample[4]: 4
This is sample[5]: 5
This is sample[6]: 6
This is sample[7]: 7
This is sample[8]:8
This is sample[9]:9
```

// Find the minimum and maximum values in an array.

```
class MinMax {
  public static void main(String[ ] args) {
    int[ ] nums = new int[10];
    int min, max;
    nums[0] = 99;
    nums[1] = -10;
    nums[2] = 100123;
    nums[3] = 18;
    nums[4] = -978;
    nums[5] = 5623;
    nums[6] = 463;
    nums[7] = -9;
    nums[8] = 287;
    nums[9] = 49;
    min = max = nums[0];
    for(int i=1; i < 10; i++) {
      if(nums[i] < min) min = nums[i];
      if(nums[i] > max) max = nums[i];
    }
    System.out.println("min and max: " + min + " " + max);
  }
}
```

// Use array initializers.

```
class MinMax2 {
  public static void main(String[] args) {
    int[ ] nums = { 99, -10, 100123, 18, -978, 5623, 463, -9, 287, 49 };
    int min, max;
    min = max = nums[0];
    for(int i=1; i < 10; i++) {
      if(nums[i] < min) min = nums[i];
      if(nums[i] > max) max = nums[i];
    }
    System.out.println("Min and max: " + min + " " + max);
  }
}
```

Multidimensional Arrays:

Two-dimensional arrays:

A two dimensional array is a list of one-dimensional array. A two dimensional array can be thought of as creating a table of data organized by row and column. An individual item of data is accessed by specifying its row and column position.

To declare a two dimensional array, we must specify two dimensions.

int[ ] [ ] table=new int[10] [20];

```
// Demonstrate a two-dimensional array.
class TwoD {
  public static void main(String[ ] args) {
    int t, i;
    int[ ][ ] table = new int[3][4];
    for(t=0; t < 3; ++t) {
      for(i=0; i < 4; ++i) {
        table[t][i] = (t*4)+i+1;
        System.out.print(table[t][i] + " ");
      }
      System.out.println( );
    }
  }
}
```

O/P:

```
1       2       3       4
5       6       7       8
9       10      11      12
```

Irregular arrays:

When allocating memory for multi dimensional arrays we need to specify only the memory for the first dimension. We can allocate the remaining dimensions separately.

```
// Manually allocate differing size second dimensions.
class Ragged {
  public static void main(String[ ] args) {
    int[ ][ ] riders = new int[7][ ];
    riders[0] = new int[10];
    riders[1] = new int[10];
    riders[2] = new int[10];
    riders[3] = new int[10];
    riders[4] = new int[10];
    riders[5] = new int[2];
    riders[6] = new int[2];
    int i, j;
    // fabricate some data
    for(i=0; i < 5; i++)
      for(j=0; j < 10; j++)
        riders[i][j] = i + j + 10;
    for(i=5; i < 7; i++)
      for(j=0; j < 2; j++)
        riders[i][j] = i + j + 10;
    System.out.println("Riders per trip during the week:");
    for(i=0; i < 5; i++) {
      for(j=0; j < 10; j++)
        System.out.print(riders[i][j] + " ");
      System.out.println( );
    }
    System.out.println( );
    System.out.println("Riders per trip on the weekend:");
    for(i=5; i < 7; i++) {
      for(j=0; j < 2; j++)
        System.out.print(riders[i][j] + " ");
      System.out.println( );
    }
  }
}
```

Initializing multi dimensional array:

A multidimensional array can be initialized by enclosing each dimension’s initialize list within its own set of braces.

// Initialize a two-dimensional array.

```
class Squares {
  public static void main(String[ ] args) {
    int[ ][ ] sqrs = {
      { 1, 1 },
      { 2, 4 },
      { 3, 9 },
      { 4, 16 },
      { 5, 25 },
      { 6, 36 },
      { 7, 49 },
      { 8, 64 },
      { 9, 81 },
      { 10, 100 }
    };
    int i, j;
    for(i=0; i < 10; i++) {
      for(j=0; j < 2; j++)
        System.out.print(sqrs[i][j] + " ");
      System.out.println( );
    }
  }
}
```

Using the length member:

Because arrays are implemented as objects, each array has associated with it a **length** instance variable that contains the number of elements that the array can hold. In other words **length** contains the size of the array.

// Use the length array member.

```
class LengthDemo {
  public static void main(String[ ] args) {
    int[ ] list = new int[10];
    int[ ] nums = { 1, 2, 3 };
    int[ ][ ] table = { // a variable-length table
      {1, 2, 3},
      {4, 5},
      {6, 7, 8, 9}
    };
    System.out.println("length of list is " + list.length);
    System.out.println("length of nums is " + nums.length);
    System.out.println("length of table is " + table.length);
    System.out.println("length of table[0] is " + table[0].length);
    System.out.println("length of table[1] is " + table[1].length);
    System.out.println("length of table[2] is " + table[2].length);
    System.out.println();
    // use length to initialize list
    for(int i=0; i < list.length; i++)
      list[i] = i * i;
    System.out.print("Here is list: ");
    // now use length to display list
    for(int i=0; i < list.length; i++)
      System.out.print(list[i] + " ");
    System.out.println( );
  }
}
```

The for-each style for loop:

A for-each style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish.

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:

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*.

Because the iteration variable receives values from the collection, *type* must be the same as (or compatible with) the elements stored in the collection.

EG: compute the sum of the values in an array:

```
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 for-each style **for** automates the preceding loop. Specifically, it eliminates the need to establish a loop counter, specify a starting and ending value, and manually index the array. Instead, it automatically cycles through the entire array, obtaining one element at a time, in sequence, from beginning to end. For example, here is the preceding fragment rewritten using a for-each version of the **for**:

```
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.

```
// Use a for-each style for loop.
class ForEach {
  public static void main(String[ ] args) {
    int[ ] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int sum = 0;
    // Use for-each style for to display and sum the values.
    for(int x : nums) {
      System.out.println("Value is: " + x);
      sum += x;
    }
    System.out.println("Summation: " + sum);
  }
}
```

There is one important point to understand about the for-each style loop. Its iteration variable is "read-only" as it relates to the underlying array. An assignment to the iteration variable has no effect on the underlying array.

```
// The for-each loop is essentially read-only.
class NoChange {
  public static void main(String[ ] args) {
    int[ ] nums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    for(int x : nums) {
      System.out.print(x + " ");
      x = x * 10; // no effect on nums
    }
    System.out.println();
    for(int x : nums)
      System.out.print(x + " ");
    System.out.println( );
  }
}
```

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

Iterating Over Multidimensional Arrays

The enhanced version of the **for** also works on multidimensional arrays.

```
// Use for-each style for on a two-dimensional array.
class ForEach2 {
  public static void main(String[ ] args) {
    int sum = 0;
    int[ ][ ] nums = new int[3][5];
    // give nums some values
    for(int i = 0; i < 3; i++)
      for(int j=0; j < 5; j++)
        nums[i][j] = (i+1)*(j+1);
    // Use for-each for loop to display and sum the values.
    for(int[] x : nums) {
      for(int y : x) {
        System.out.println("Value is: " + y);
        sum += y;
      }
    }
    System.out.println("Summation: " + sum);
  }
}
```

O/P:
```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 2
Value is: 4
Value is: 6
Value is: 8
Value is: 10
Value is: 3
Value is: 6
Value is: 9
Value is: 12
Value is: 15
Summation: 90
```

```
// Search an array using for-each style for.
import java.util.Scanner;
public class Search {
             public static void main(String[] args) {
        int[] nums = { 6, 8, 3, 7, 5, 6, 1, 4 };
        int val;
        Scanner a=new Scanner(System.in);
        System.out.println("Enter a number to search");
        val=a.nextInt();
        boolean found = false;
        // Use for-each style for to search nums for val.
        for(int x : nums) {
          if(x == val) {
            found = true;
            break;
          }
        }
        if(found)
          System.out.println("Value found!");
        else
            System.out.println("Value not found!");
       }
      }
```

O/P:
```
Enter a number to search
8
Value found!



Enter a number to search
10
Value not found!
```

STRINGS:

One of the most important data type in java is String. String defines and supports character sting.

In java strings are objects

Constructing strings:

String str= new String("HELLO");

This creates a String object called str that contains the character string "HELLO".

A string can be constructed from another string

String str2= new String(str);

Another easy way to create a string is

String str="Java strings are powerful";

```
// Introduce String.
class StringDemo {
  public static void main(String[ ] args) {
    // declare strings in various ways
    String str1 = new String("Java strings are objects.");
    String str2 = "They are constructed various ways.";
    String str3 = new String(str2);
    System.out.println(str1);
    System.out.println(str2);
    System.out.println(str3);
  }
}
```

Operating on strings:

The **String** class contains several methods that operate on strings.

boolean equals(str)	Returns true if the invoking string contains the same character sequence as str
int length( )	Returns the number of characters in the string
char charAt(index)	Returns the character at the index specified by index
int compareTo(str)	Returns less than zero if the invoking string is less than str, greater than zero if the the invoking string is greater than str, and zero if the strings are equal
int indexOf(str)	Searches the invoking string for the substring specified by str. Returns the index of the first match or -1 on failure
int lastIndexOf(str)	Searches the invoking string for the substring specified by str. Returns the index of the last match or -1 on failure

```
// Some String operations.
class StrOps {
  public static void main(String[] args) {
    String str1 =  "When it comes to Web programming, Java is #1.";
    String str2 = new String(str1);
    String str3 = "Java strings are powerful.";
    int result, idx;
    char ch;
    System.out.println("Length of str1: " + str1.length());
    for(int i=0; i < str1.length(); i++)               // display str1, one char at a time.
      System.out.print(str1.charAt(i));
    System.out.println();
    if(str1.equals(str2))
      System.out.println("str1 equals str2");
    else
      System.out.println("str1 does not equal str2");
    if(str1.equals(str3))
      System.out.println("str1 equals str3");
    else
      System.out.println("str1 does not equal str3");
    result = str1.compareTo(str3);
    if(result == 0)
      System.out.println("str1 and str3 are equal");
    else if(result < 0)
      System.out.println("str1 is less than str3");
    else
      System.out.println("str1 is greater than str3");
    // assign a new string to str2
    str2 = "One Two Three One";
    idx = str2.indexOf("One");
    System.out.println("Index of first occurrence of One: " + idx);
    idx = str2.lastIndexOf("One");
    System.out.println("Index of last occurrence of One: " + idx);
  }
}
```

Array of strings:

```
// Demonstrate String arrays.
class StringArrays {
  public static void main(String[ ] args) {
    String[] strs = { "This", "is", "a", "test." };
    System.out.println("Original array: ");
    for(String s : strs)
      System.out.print(s + " ");
    System.out.println("\n");
    // change a string in the array
    strs[1] = "was";
    strs[3] = "test, too!";
    System.out.println("Modified array: ");
    for(String s : strs)
      System.out.print(s + " ");
  }
}
```

```
// Use substring().
class SubStr {
  public static void main(String[] args) {
    String orgstr = "Java makes the Web move.";
    // construct a substring
    String substr = orgstr.substring(5, 18);
    System.out.println("orgstr: " + orgstr);
    System.out.println("substr: " + substr);
  }
}
```

```
// Use a string to control a switch statement.
class StringSwitch {
  public static void main(String[ ] args) {
    String command = "cancel";
    switch(command) {
      case "connect":      System.out.println("Connecting");
                            // ...
                           break;
      case "cancel":        System.out.println("Canceling");
                           // ...
                            break;
      case "disconnect":  System.out.println("Disconnecting");
                           // ...
                           break;
      default:                 System.out.println("Command Error!");
                           break;
    }
  }
}
```

Using command line arguments:

```
// Display all command-line information.
class CLDemo {
  public static void main(String[ ] args) {
    System.out.println("There are " + args.length + " command-line arguments.");
    System.out.println("They are: ");
    for(int i=0; i<args.length; i++)
      System.out.println("arg[" + i + "]: " + args[i]);
  }
}
```

STRING HANDLING:

The **String** class is packaged in **java.lang**. Thus it is automatically available to all programs.

String objects can be constructed in a number of ways, making it easy to obtain a string when needed.

String Constructors:

// Demonstrate several String constructors.

```
class StringConsDemo {
  public static void main(String[ ] args) {
    char[ ] digits = new char[16];
    // Create an array that contains the digits 0 through 9
    // plus the hexadecimal values A through F.
     for(int i=0; i < 16; i++) {
      if(i < 10) digits[i] = (char) ('0'+i);
      else digits[i] = (char) ('A' + i - 10);
    }
    // Create a string that contains all of the array.
    String digitsStr = new String(digits);
    System.out.println(digitsStr);
    // Create a string the contains a portion of the array.
    String nineToTwelve = new String(digits, 9, 4);
    System.out.println(nineToTwelve);
    // Construct a string from a string.
    String digitsStr2 = new String(digitsStr);
    System.out.println(digitsStr2);
    // Now, create an empty string.
    String empty = new String();
    // This will display nothing:
    System.out.println("Empty string: " + empty);
  }
}
```

String Concatenation

In general, Java does not allow operators to be applied to **String** objects. The one exception to this rule is the **+** operator, which concatenates two strings, producing a **String** object as the result.

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println (s);
```

This displays the string "He is 9 years old."

String Concatenation with Other Data Types

You can concatenate strings with other types of data. For example, consider this slightly different version of the earlier example:

```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println (s);
```

In this case, **age** is an **int** rather than another **String**, but the output produced is the same as before. This is because the **int** value in **age** is automatically converted into its string representation within a **String** object.

One practical use of string concatenation is found when you are creating very long strings. Instead of letting long strings wrap around within your source code, you can break them into smaller pieces, using the **+** to concatenate them. Here is an example:

```
// Using concatenation to prevent long lines.
class ConCat {
public static void main(String args[ ]) {
String longStr = "This could have been " +
"a very long line that would have " +
"wrapped around. But string concatenation " +
"prevents this.";
System.out.println(longStr);
}
}
```

Character Extraction

The **String** class provides a number of ways in which characters can be extracted from a **String** object

charAt( )

To extract a single character from a **String**, you can refer directly to an individual character via the **charAt( )** method. It has this general form:

char charAt(int *index*)

// Demonstrate charAt( ) and length( ).

```
class CharAtAndLength {
  public static void main(String[] args) {
    String str = "Programming is both art and science.";
    // Cycle through all characters in the string.
    for(int i=0; i < str.length(); i++)
      System.out.print(str.charAt(i) + " ");
    System.out.println();
  }
}
```

getChars( )

If you need to extract more than one character at a time, you can use the **getChars( )** method. It has this general form:

void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

// getChars( )

```
class GetCharsDemo {
  public static void main(String[] args) {
    String str = "Programming is both art and science.";
    int start = 15;
    int end = 23;
    char[ ] buf = new char[end - start];
    str.getChars(start, end, buf, 0);
    System.out.println(buf);
  }
}
```

String Comparison

The **String** class includes a number of methods that compare strings or substrings within strings.

equals( ) and equalsIgnoreCase( )

To compare two strings for equality, use **equals( )**. It has this general form:

boolean equals(Object *str*)

Here, *str* is the **String** object being compared with the invoking **String** object. It returns **true** if the strings contain the same characters in the same order, and **false** otherwise.

The comparison is case-sensitive.

To perform a comparison that ignores case differences, call **equalsIgnoreCase( )**.

When it compares two strings, it considers **A-Z** to be the same as **a-z**. It has this general form:

boolean equalsIgnoreCase(String *str*)

Here, *str* is the **String** object being compared with the invoking **String** object. It, too, returns **true** if the strings contain the same characters in the same order, and **false** otherwise.

// Demonstrate equals() and equalsIgnoreCase().

```
class EqualityDemo {
  public static void main(String[] args) {
    String str1 = "table";
    String str2 = "table";
    String str3 = "chair";
    String str4 = "TABLE";
    if(str1.equals(str2))
      System.out.println(str1 + " equals " + str2);
    else
      System.out.println(str1 + " does not equal " + str2);
    if(str1.equals(str3))
      System.out.println(str1 + " equals " + str3);
    else
      System.out.println(str1 + " does not equal " + str3);
    if(str1.equals(str4))
      System.out.println(str1 + " equals " + str4);
    else
      System.out.println(str1 + " does not equal " + str4);
    if(str1.equalsIgnoreCase(str4))
      System.out.println("Ignoring case differences, " + str1 +" equals " + str4);
    else
      System.out.println(str1 + " does not equal " + str4);
  }
}
```

O/P:

```
table equals table
table does not equal chair
table does not equal TABLE
Ignoring case differences, table equals TABLE
```

equals( ) Versus = =

It is important to understand that the equals( ) method and the == operator perform two different operations. The equals( ) method compares the characters inside a String object. The == operator compares two object references to see whether they refer to the same instance.

```
// equals( ) vs = =
class EqualsNotEqualTo {
        public static void main(String args[ ]) {
                String s1 = "Hello";
                String s2 = new String(s1);
                System.out.println(s1 + " equals " + s2 + " -> " +
                s1.equals(s2));
                System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
        }
}
```

The variable s1 refers to the String instance created by “Hello”. The object referred to by s2 is created with s1 as an initializer. Thus, the contents of the two String objects are identical, but they are distinct objects. This means that s1 and s2 do not refer to the same objects and are, therefore, not = =, as is shown here by the output of the preceding example:

```
Hello equals Hello -> true
Hello == Hello -> false
```

regionMatches( )

The **regionMatches( )** method compares a specific region inside a string with another specific region in another string. Here are the general forms for two methods:

boolean regionMatches(int *startIndex,* String *str2,*

int *str2StartIndex,* int *numChars*)

boolean regionMatches(boolean *ignoreCase,*

int *startIndex,* String *str2,*

int *str2StartIndex,* int *numChars*)

```
// Demonstrate RegionMatches.
class CompareRegions {
  public static void main(String[] args) {
    String str1 = "Standing at river's edge.";
    String str2 = "Running at river's edge.";
    if(str1.regionMatches(9, str2, 8, 12))
      System.out.println("Regions match.");
    if(!str1.regionMatches(0, str2, 0, 12))
      System.out.println("Regions do not match.");
  }
}
```

O/P:

```
Regions match.
Regions do not match.
```

startsWith( ) and endsWith( )

The **startsWith( )** method determines whether a given **String** begins with a specified string. Conversely, **endsWith( )** determines whether the **String** in question ends with a specified string. They have the following general forms:

boolean startsWith(String *str*)

boolean endsWith(String *str*)

A second form of **startsWith( )**, shown here, lets you specify a starting point:

boolean startsWith(String *str*, int *startIndex*)

compareTo( ) and compareToIgnoreCase( )

It has this general form:

int compareTo(String *str*)

Here, *str* is the **String** being compared with the invoking **String**. The result of the comparison is returned and is interpreted as shown here:

Value	Meaning
Less than zero	The invoking string is less than str
Greater than zero	The invoking string is greater than str
Zero	The two strings are equal.

If you want to ignore case differences when comparing two strings, use **compareToIgnoreCase( )**, as shown here:

int compareToIgnoreCase(String *str*)

substring( )

You can extract a substring using **substring( )**. It has two forms. The first is

String substring(int *startIndex*)

Here, *startIndex* specifies the index at which the substring will begin. This form returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string.

The second form of **substring( )** allows you to specify both the beginning and ending index of the substring:

String substring(int *startIndex*, int *endIndex*)

Here, *startIndex* specifies the beginning index, and *endIndex* specifies the stopping point

replace( )

The **replace( )** method has two forms. The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:

String replace(char *original*, char *replacement*)

Here, *original* specifies the character to be replaced by the character specified by *replacement*. The resulting string is returned. For example,

String s = "Hello".replace('l', 'w');

puts the string "Hewwo" into **s**.

The second form of **replace( )** replaces one character sequence with another. It has this general form:

String replace(CharSequence *original,* CharSequence *replacement*)

trim( )

The **trim( )** method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:

String trim( )

Eg: String str="   gamma   ";

After   str=str.trim( );

Str will contain only the string"gamma"

Changing the Case of Characters Within a String

The method **toLowerCase( )** converts all the characters in a string from uppercase to lowercase. The **toUpperCase( )** method converts all the characters in a string from lowercase to uppercase. Nonalphabetical characters, such as digits, are unaffected. Here are the simplest forms of these methods:

String toLowerCase( )
String toUpperCase( )

// Demonstrate toUpperCase() and toLowerCase().

```
class ChangeCase {
  public static void main(String[] args)
  {
    String str = "This is a test.";
    System.out.println("Original: " + str);
    String upper = str.toUpperCase();
    String lower = str.toLowerCase();
    System.out.println("Uppercase: " + upper);
    System.out.println("Lowercase: " + lower);
  }
}
```

O/P:

```
Original: This is a test.
Uppercase: THIS IS A TEST.
Lowercase: this is a test.
```

String represents fixed-length, immutable character sequences.

In contrast, **StringBuffer** represents growable and writable character sequences. **StringBuffer** may have characters and substrings inserted in the middle or appended to the end.

StringBuilder is identical to **StringBuffer** except for one important difference: it is not synchronized, which means that it is not thread-safe. The advantage of **StringBuilder** is faster performance.