UNIT II

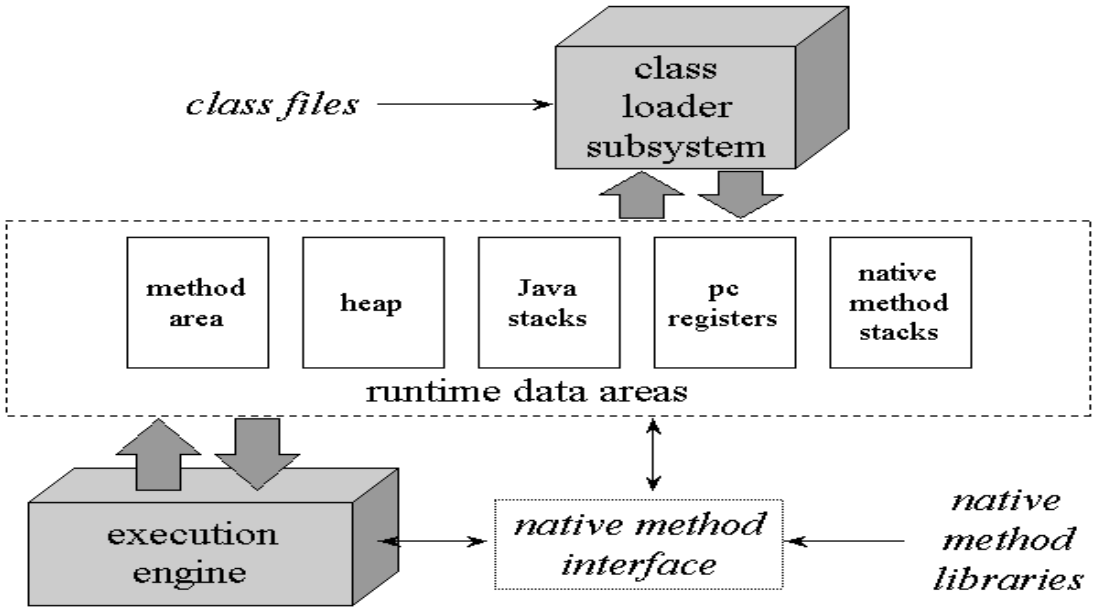
Java Basics History of Java, Java buzzwords, datatypes, variables, scope and life time of variables, arrays, operators, expressions, control statements, type conversion and casting, simple java program, concepts of classes, objects, constructors, methods, access control, this keyword, garbage collection, overloading methods and constructors, parameter passing, recursion, nested and inner classes, exploring string class

HISTORY OF JAVA

- In 1990, Sun Micro Systems Inc. (US) was conceived a project to develop software for consumer electronic devices that could be controlled by a remote. This project was called Stealth Project but later its name was changed to Green Project.
- In January 1991, Project Manager James Gosling and his team members Patrick Naughton, Mike Sheridan, Chris Wrath, and Ed Frank met to discuss about this project.
- Gosling thought C and C++ would be used to develop the project. But the problem he faced with them is that they were system dependent languages. The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target and could not be used on various processors, which the electronic devices might use.
- James Gosling with his team started developing a new language, which was completely system independent. This language was initially called **OAK**. Since this name was registered by some other company, later it was changed to **Java**.
- James Gosling and his team members were consuming a lot of coffee while developing this language. Good quality of coffee was supplied from a place called "Java Island'. Hence they fixed the name of the language as Java. The symbol for Java language is cup and saucer.
- Sun formally announced Java at Sun World conference in 1995. On January 23rd 1996, JDK1.0 version was released.

THE JAVA VIRTUAL MACHINE

- Java Virtual Machine (JVM) is the heart of entire Java program execution process. First of all, the .java program is converted into a .class file consisting of byte code instructions by the java compiler at the time of compilation. Remember, this java compiler is outside the JVM. This .class file is given to the JVM. Following figure shows the architecture of Java Virtual Machine.



- In JVM, there is a module (or program) called class loader sub system, which performs the following instructions:
- · First of all, it loads the .class file into memory.
- · Then it verifies whether all byte code instructions are proper or not. If it finds any instruction suspicious, the execution is rejected immediately.
-
- If the byte instructions are proper, then it allocates necessary memory to execute the program. This memory is divided into 5 parts, called run time data areas, which contain the data and results while running the program. These areas are as follows:
-
- o **Method area:** Method area is the memory block, which stores the class code, code of the variables and code of the methods in the Java program. (Method means functions written in a class).
-
- o **Heap:** This is the area where objects are created. Whenever JVM loads a class, method and heap areas are immediately created in it.
-
- o **Java Stacks:** Method code is stored on Method area. But while running a method, it needs some more memory to store the data and results. This memory is allotted on Java Stacks. So, Java Stacks are memory area where Java methods are executed. While executing methods, a separate frame will be created in the Java Stack, where the method is executed. JVM uses a separate thread (or process) to execute each method.

- o **PC (Program Counter) registers:** These are the registers (memory areas), which contain memory address of the instructions of the methods. If there are 3 methods, 3 PC registers will be used to track the instruction of the methods.
-
- o **Native Method Stacks:** Java methods are executed on Java Stacks. Similarly, native methods (for example C/C++ functions) are executed on Native method stacks. To execute the native methods, generally native method libraries (for example C/C++ header files) are required. These header files are located and connected to JVM by a program, called Native method interface.

JAVA BUZZWORDS

■ Simple
■ Secure
■ Portable
■ Object-oriented
■ Robust
■ Multithreaded
■ Architecture-neutral
■ Interpreted
■ High performance
■ Distributed
■ Dynamic

Simple

Java is a small and simple language. Java does not use pointers, pre-processor header files, goto statement and many other. It also eliminates operator overloading and multiple inheritance. Java inherits the C/C++ syntax and many of the object oriented features of C++.

Secure

Security becomes an important issue for a language that is used for programming on Internet. Every time when you download a "normal program", here is a risk of viral infection. When we use a java compatible web browser, we can safely download Java applets without fear of viral infection. Java achieves this protection by confining a Java program to the Java execution environment and not allowing it access to other parts of the computer.

Portable

Java programs can be easily moved from one computer system to another, anywhere and anytime. This is the reason why Java has become a popular language for programming on Internet.

Object-Oriented

Java is a true object oriented language. Almost everything in java is an object. All program code and data reside within objects and classes Java comes with an extensive set of classes, arranged in packages, that we can use in our programs by inheritance. The object model in java is simple and easy to extend.

Robust

Java is a robust language. It provides many safeguards to ensure reliable code. To gaon reliability, Java restricts in few key areas

i. to force you to find mistakes early in program development.
ii. java frees you from having to worry about many of the most common causes of programming errors.

Java is a strictly typed language. It checks your code at compile time.
Two main reasons for program failure are:

1. Memory management mistakes and
2. Mishandled exceptional conditions (i.e runtime errors)

Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer must manually allocates and free all dynamic memory. This sometime leads to problems, because programmers will either forget to free memory that has been previously allocated or try to free some memory that another part of there is still using. Java virtually eliminates these problems by managing allocation and de allocation. De allocation is completely automatic, because java provides garbage collection for unused objects.
Exceptional conditions often arise in situations such as division by zero or file not found etc.. Java helps in this area by providing object oriented exception handling.

Multithreaded

Multithreaded means handling multiple tasks simultaneously. This means that we need not wait for the application to finish one task before beginning another. To accomplish this, java supports multithreaded programming which allows to write programs that do many things simultaneously.

Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. Operating system

upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. Java Virtual Machine solves this problem. The goal is "write once; run anywhere, any time, forever."

Interpreted and High Performance

Java performance is impressive for an interpreted language, mainlt deu to the use of byte code. This code can be interpreted on any system that provides a JVM. Java was designed to perform well on very low power CPUs.

Distributed

Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. The original version of Java (Oak) included features for intraaddress-space messaging. This allowed objects on two different computers to execute procedures remotely. Java revived these interfaces in a package called Remote Method Invocation (RMI). This feature brings an unparalleled level of abstraction to client/ server programming.

Dynamic

Java programs carry with them substantial amounts of 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 and expedient manner.

DATATYPES

Java defines eight simple (or elemental) types of data: **byte**, **short**, **int**, **long**, **char**, **float**,**double**, and **boolean**. These can be put in four groups:

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

Integer Data Types: These data types store integer numbers

Data Type	Memory size	Range
Byte	1 byte	-128 to 127
Short	2 bytes	-32768 to 32767
Int	4 bytes	-2147483648 to 2147483647
Long	8 bytes	-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;
// approximate speed of light in miles per second
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.");
}
}
```

This program generates the following output:

In 1000 days light will travel about 16070400000000 miles.

Float Data Types: These data types handle floating point numbers

Data Type	Memory size	Range
Float	4 bytes	-3.4e38 to 3.4e38
Double	8 bytes	-1.7e308 to 1.7e308

```
// Compute the area of a circle.
class Area {
public static void main(String args[]) {
double pi, r, a;
r = 10.8; // radius of circle
pi = 3.1416; // pi, approximately
a = pi * r * r; // compute area
System.out.println("Area of circle is " + a);
}
}
```

Character Data Type: This data type represents a single character. char data type in java uses two bytes of memory also called Unicode system. Unicode is a specification to include alphabets of all international languages into the character set of java.

Data Type	Memory size	Range
Char	2 bytes	0 to 65535

Here is a program that demonstrates **char** variables:

```
// Demonstrate char data type.
class CharDemo {
public static void main(String args[]) {
char ch1, ch2;
ch1 = 88; // code for X
ch2 = 'Y';
System.out.print("ch1 and ch2: ");
System.out.println(ch1 + " " + ch2);
}
}
```

This program displays the following output:

```
ch1 and ch2: X Y
```

Notice that **ch1** is assigned the value 88, which is the ASCII (and Unicode) value that corresponds to the letter *X*. As mentioned, the ASCII character set occupies the first 127 values in the Unicode character set. For this reason, all the "old tricks" that you have used with characters in the past will work in Java, too. Even though **char**s are not integers, in many cases you can operate on them as if they were integers. This allows you to add two characters together, or to increment the value of a character variable. For example, consider the following program:

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

The output generated by this program is shown here:

```
ch1 contains X
ch1 is now Y
```

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

Boolean Data Type: can handle truth values either true or false
e.g.:- boolean response = true;

Here is a program that demonstrates the **boolean** type:

```
// Demonstrate boolean values.
class BoolTest {
public static void main(String args[]) {
boolean b;
b = false;
System.out.println("b is " + b);
b = true;
System.out.println("b is " + b);
// a boolean value can control the if statement
if(b) System.out.println("This is executed.");
b = false;
if(b) System.out.println("This is not executed.");
// outcome of a relational operator is a boolean value
System.out.println("10 > 9 is " + (10 > 9));
}
}
```

The output generated by this program is shown here:

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

VARIABLES

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime.

Declaring a Variable

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

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

The *type* is one of Java's atomic types, or the name of a class or interface. The *identifier* is the name of the variable.

Eg : int n;

Eg: int n=10, m=20;

Dynamic Initialization

Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.

```
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);
}
}
```

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 you start a new block, you are creating a new scope.
- ✓ A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.
- ✓ Most other computer languages define two general categories of scopes: global and local.
- ✓ In Java, the two major scopes are those defined by a class and those defined by a method.
- ✓ The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope.
- ✓ As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification.
- ✓ Indeed, the scope rules provide the foundation for encapsulation.
- ✓ Scopes can be nested. The outer scope encloses the inner scope This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.

```
// Demonstrate block scope.
class Scope {
public static void main(String args[]) {
int x;
```

```
x = 10;
if(x == 10) {
int y = 20;
// 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);
}
}
```

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

```
// Demonstrate lifetime of a variable.
class LifeTime {
public static void main(String args[]) {
int x;
for(x = 0; x < 3; x++) {
int y = -1; // y is initialized each time block is entered
System.out.println("y is: " + y); // this always prints -1
y = 100;
System.out.println("y is now: " + y);
}
}
}
```

The output generated by this program is shown here:

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

TYPE CONVERSION AND CASTING

- ✓ Type casting is 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.
- ✓ For instance, there is no conversion defined from **double** to **byte**. To do so, you must use a *cast*, which performs an explicit conversion between incompatible types.

Java's Automatic Conversions

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

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

When these two conditions are met, a ***widening conversion*** takes place.

For example, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required.

Casting Incompatible Types

For example, what if you want to assign an **int** value to a **byte** variable? This conversion will not be performed automatically, because a **byte** is smaller than an **int**. This kind of conversion is sometimes called a ***narrowing conversion***, since you are explicitly making the value narrower so that it will fit into the target type. To create a conversion between two incompatible types, you must use a cast. A ***cast*** is simply an explicit type conversion. It has this general form:

(*target-type*) *value*

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

```
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);
}
}
```

```
Conversion of int to byte.
i and b 257 1
Conversion of double to int.
d and i 323.142 323
Conversion of double to byte.
d and b 323.142 67
```

ARRAYS

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

One-Dimensional Arrays

The general form of a onedimensional array declaration is

type var-name[ ];

Here, *type* declares the base type of the array.

eg: int month_days[];
you must allocate one using **new** and assign it to **month_days**. **new** is a special operator that allocates memory.

The general form of **new** as it applies to one-dimensional arrays appears as follows:

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

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

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

the following code creates an initialized array of integers:

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

Multidimensional Arrays

In Java, *multidimensional arrays* are actually arrays of arrays.
To declare a multidimensional array variable, specify each additional index using another set of square brackets.

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

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

```
// 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
```

```
// Initialize a two-dimensional array.
class Matrix {
public static void main(String args[]) {
double m[][] = {
{ 0*0, 1*0, 2*0, 3*0 },
{ 0*1, 1*1, 2*1, 3*1 },
{ 0*2, 1*2, 2*2, 3*2 },
{ 0*3, 1*3, 2*3, 3*3 }
};
int i, j;
for(i=0; i<4; i++) {
for(j=0; j<4; j++)
System.out.print(m[i][j] + " ");
System.out.println();
}
}
}
```

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

```
0.0 0.0 0.0 0.0
0.0 1.0 2.0 3.0
0.0 2.0 4.0 6.0
0.0 3.0 6.0 9.0
```

Alternative Array Declaration Syntax

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

type[ ] *var-name;*

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

```
int al[] = new int[3];
int[] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

This alternative declaration form is included as a convenience, and is also useful when specifying an array as a return type for a method.

OPERATORS

An operator is a symbol that performs an operation. An operator acts on variables called operands.

Arithmetic operators: These operators are used to perform fundamental operations like addition, subtraction, multiplication etc.

Operator	Meaning	Example	Result
+	Addition	3 + 4	7
-	Subtraction	5 - 7	-2
*	Multiplication	5 * 5	25
/	Division (gives quotient)	14 / 7	2
%	Modulus (gives remainder)	20 % 7	6

Assignment operator: This operator (=) is used to store some value into a variable.

Simple Assignment	Compound Assignment
x = x + y	x += y
x = x – y	x -= y
x = x * y	x *= y
x = x y	x /= y

Unary operators: As the name indicates unary operator's act only on one operand.

Operator	Meaning	Example	Explanation
-	Unary minus	j = -k;	k value is negated and stored into j
++	Increment Operator	b++; ++b;	b value will be incremented by 1 (called as post incrementation) b value will be incremented by 1 (called as pre incrementation)
--	Decrement Operator	b--; --b;	b value will be decremented by 1 (called as post decrementation) b value will be decremented by 1 (called as pre decrementation)

Relational operators: These operators are used for comparison purpose.

Operator	Meaning	Example
==	Equal	x == 3
!=	Not equal	x != 3
<	Less than	x < 3
>	Greater than	x > 3
<=	Less than or equal to	x <= 3

Logical operators: Logical operators are used to construct compound conditions. A compound condition is a combination of several simple conditions.

Operator	Meaning	Example	Explanation
&&	and operator	if(a>b && a>c) System.out.print("yes");	If a value is greater than b and c then only yes is displayed
||	or operator	if(a==1 || b==1) System.out.print("yes");	If either a value is 1 or b value is 1 then yes is displayed
!	not operator	if(!(a==0)) System.out.print("yes");	If a value is not equal to zero then only yes is displayed

Bitwise operators: These operators act on individual bits (0 and 1) of the operands. They act only on integer data types, i.e. byte, short, long and int.

Operator	Meaning	Explanation
&	Bitwise AND	Multiplies the individual bits of operands
|	Bitwise OR	Adds the individual bits of operands
^	Bitwise XOR	Performs Exclusive OR operation
<<	Left shift	Shifts the bits of the number towards left a specified number of positions
>>	Right shift	Shifts the bits of the number towards right a specified number of positions and also preserves the sign bit.
>>>	Zero fill right shift	Shifts the bits of the number towards right a specified number of positions and it stores 0 (Zero) in the sign bit.
~	Bitwise complement	Gives the complement form of a given number by changing 0's as 1's and vice versa.

Ternary Operator or Conditional Operator (? :):

This operator is called ternary because it acts on 3 variables. The syntax for this operator is:
Variable = Expression1? Expression2: Expression3;

First Expression1 is evaluated. If it is true, then Expression2 value is stored into variable otherwise Expression3 value is stored into the variable.

e.g.: max = (a>b) ? a: b;

Program 1: Write a program to perform arithmetic operations

```
//Addition of two numbers
class AddTwoNumbers
{ public static void mian(String args[])
{ int i=10, j=20;
System.out.println("Addition of two numbers is : " + (i+j));
System.out.println("Subtraction of two numbers is : " + (i-j));
System.out.println("Multiplication of two numbers is : " + (i*j));
System.out.println("Quotient after division is : " + (i/j) );
System.out.println("Remainder after division is : " +(i%j) );
}
}
```

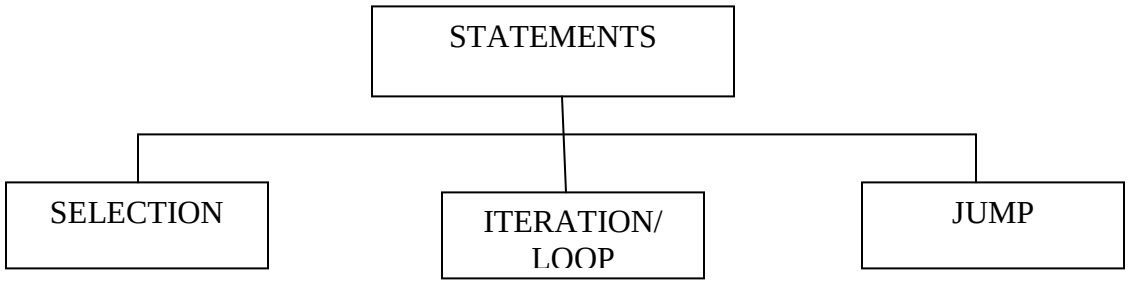
Program 2: Write a program to perform Bitwise operations

```
//Bitwise Operations
class Bits
```

```
{ public static void main(String args[])
{ byte x,y;
x=10;
y=11;
System.out.println ("~x="+(~x));
System.out.println ("x & y="+(x&y));
System.out.println ("x | y="+(x|y));
System.out.println ("x ^ y="+(x^y));
System.out.println ("x<<2="+(x<<2));
System.out.println ("x>>2="+(x>>2));
System.out.println ("x>>>2="+(x>>>2));
}
}
```

CONTROL STATEMENTS

Statements are divided into three groups:



Java's Selection Statements

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

if

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

if (*condition*) *statement1*;
else *statement2*;

Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a *block*). The *condition* is any expression that returns a **boolean** value.

The **else** clause is optional.
The **if** works like this: If the *condition* is true, then *statement1* is executed. Otherwise, *statement2* (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

```
int a, b;
// ...
if(a < b) a = 0;
else b = 0;
```

Nested ifs

A *nested* **if** is an **if** statement that is the target of another **if** or **else**. Nested **if**s are very common in programming. 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**. Here is an example:

```
if(i == 10) {
if(j < 20) a = b;
if(k > 100) c = d; // this if is
else a = c; // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

The 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;
```

switch

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

Here is the general form of a **switch** statement:

```
switch (expression) {
case value1:
// statement sequence
break;
case value2:
// statement sequence
break;
...
case valueN:
// statement sequence
break;
default:
// default statement sequence
}
```

```
class SampleSwitch {
public static void main(String args[]) {
for(int i=0; i<6; i++)
switch(i) {
case 0:
System.out.println("i is zero.");
break;
case 1:
System.out.println("i is one.");
break;
case 2:
System.out.println("i is two.");
break;
case 3:
System.out.println("i is three.");
break;
default:
System.out.println("i is greater than 3.");
}
}
}
```

The output produced by this program is shown here:

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

Iteration Statements

Java's iteration statements are

for
while
do-while

These statements are used to repeat same set of instructions specified number of times called loops. A loop repeatedly executes the same set of instructions until a termination condition is met.

- **while Loop:** while loop repeats a group of statements as long as condition is true. Once the condition is false, the loop is terminated. In while loop, the condition is tested first; if it is true, then only the statements are executed. while loop is called as entry control loop.

Syntax: while (condition)
{
statements;
}

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural
{ public static void main(String args[])
{ int i=1;
while (i <= 20)
{ System.out.print (i + "\t");
i++;
}
}
}
```

do…while Loop: do…while loop repeats a group of statements as long as condition is true. In do...while loop, the statements are executed first and then the condition is tested. do…while loop is also called as exit control loop.

Syntax: do
{
statements;
} while (condition);

Program: Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural
```

```
{ public static void main(String args[])
{ int i=1;
do
{ System.out.print (i + "\t");
i++;
} while (i <= 20);
}
}
```

for Loop: The for loop is also same as do…while or while loop, but it is more compact syntactically. The for loop executes a group of statements as long as a condition is true.

Syntax: for (expression1; expression2; expression3)
{ statements;
}
Here, expression1 is used to initialize the variables, expression2 is used for condition checking and expression3 is used for increment or decrement variable value.

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural
{ public static void main(String args[])
{ int i;
for (i=1; i<=20; i++)
System.out.print (i + "\t");
}
}
```

JUMP STATEMENTS

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

- **break:**
  - break can be used inside a loop to come out of it.
  - break can be used inside the switch block to come out of the switch block.
  - break can be used in nested blocks to go to the end of a block. Nested blocks represent a block written within another block.
  -

Syntax: break; (or) break label; //here label represents the name of the block.

Program : Write a program to use break as a civilized form of goto.

```
//using break as a civilized form of goto
class BreakDemo
{ 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");
}
}
}
```

continue: This statement is useful to continue the next repetition of a loop/ iteration. When continue is executed, subsequent statements inside the loop are not executed.

Syntax: continue;

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural
{ public static void main (String args[])
{ int i=1;
while (true)
{ System.out.print (i + "\t");
i++;
if (i <= 20 )
continue;
else
break;
}
}
}
```

return statement:

- return statement is useful to terminate a method and come back to the calling method.
- return statement in main method terminates the application.
- return statement can be used to return some value from a method to a calling method.

-

Syntax: return;
(or)
return value; // value may be of any type

Program : Write a program to demonstrate return statement.

```
//Demonstrate return
class ReturnDemo
{ public static void main(String args[])
{ boolean t = true;
System.out.println ("Before the return");
if (t)
return;
System.out.println ("This won't execute");
} }
```

A FIRST SIMPLE PROGRAM

- As all other programming languages, Java also has a structure.
- The first line of the C/C++ program contains include statement. For example, <stdio.h> is the header file that contains functions, like printf (), scanf () etc. So if we want to use any of these functions, we should include this header file in C/ C++ program.
- Similarly in Java first we need to import the required packages. By default java.lang.* is imported. Java has several such packages in its library. A package is a kind of directory that contains a group of related classes and interfaces. A class or interface contains methods.
- Since Java is purely an Object Oriented Programming language, we cannot write a Java program without having at least one class or object. So, it is mandatory to write a class in Java program. We should use class keyword for this purpose and then write class name.
- In C/C++, program starts executing from main method similarly in Java, program starts executing from main method. The return type of main method is void because program starts executing from main method and it returns nothing.
- Since Java is purely an Object Oriented Programming language, without creating an object to a class it is not possible to access methods and members of a class. But main method is also a method inside a class, since program execution starts from main method we need to call main method without creating an object.
- Static methods are the methods, which can be called and executed without creating objects.
- Since we want to call main () method without using an object, we should declare main ()

Sample Program:

```
class Sample
{
public static void main(String args[])
{
System.out.print ("Hello world");
}
}
```

- JVM calls main () method using its Classname.main () at the time of running the program. JVM is a program written by Java Soft people (Java development team) and main () is the method written by us. Since, main () method should be available to the JVM, it should be declared as public. If we don't declare main () method as public, then it doesn't make itself available to JVM and JVM cannot execute it.
- JVM always looks for main () method with String type array as parameter otherwise JVM cannot recognize the main () method, so we must provide String type array as parameter to main () method.
- A class code starts with a {and ends with a}. A class or an object contains variables and methods (functions). We can create any number of variables and methods inside the class.
- This is our first program, so we had written only one method called main ().
- Our aim of writing this program is just to display a string "Hello world".
- In Java, print () method is used to display something on the monitor. A method should be called by using objectname.methodname (). So, to call print () method, create an object to PrintStream class then call objectname.print () method.
- An alternative is given to create an object to PrintStream Class i.e. System.out. Here, System is the class name and out is a static variable in System class. out is called a field in System class. When we call this field a PrintStream class object will be created internally. So, we can call print() method as: System.out.print ("Hello world"); println () is also a method belonging to PrintStream class. It throws the cursor to the next line after displaying the result.
- In the above Sample program System and String are the classes present in java.lang package.

Introduction to OOPs

Languages like Pascal, C, FORTRAN, and COBOL are called procedure oriented programming languages. Since in these languages, a programmer uses procedures or functions to perform a task. When the programmer wants to write a program, he will first divide the task into separate sub tasks, each of which is expressed as functions/ procedures. This approach is called procedure oriented approach.

The languages like C++ and Java use classes and object in their programs and are called Object Oriented Programming languages. The main task is divided into several modules and these are represented as classes. Each class can perform some tasks for which several methods are written in a class. This approach is called Object Oriented approach.

Difference between Procedure Oriented Programming and OOP:

Procedure Oriented Programming	Object Oriented Programming
1. Main program is divided into small parts depending on the functions.	1. Main program is divided into small object depending on the problem.
2. The Different parts of the program connect with each other by parameter passing & using operating system.	2. Functions of object linked with object using message passing.
3. Every function contains different data.	3. Data & functions of each individual object act like a single unit.
4. Functions get more importance than data in program.	4. Data gets more importance than functions in program.
5. Most of the functions use global data.	5. Each object controls its own data.
6. Same data may be transfer from one function to another	6. Data does not possible transfer from one object to another.
7. There is no perfect way for data hiding.	7. Data hiding possible in OOP which prevent illegal access of function from outside of it. This is one of the best advantages of OOP also.
8. Functions communicate with other functions maintaining as usual rules.	8. One object link with other using the message passing.
9. More data or functions can not be added with program if necessary. For this purpose full program need to be change.	9. More data or functions can be added with program if necessary. For this purpose full program need not to be change.
10. To add new data in program user should be ensure that function allows it.	10. Message passing ensure the permission of accessing member of an object from other object.
11. Top down process is followed for program design.	11. Bottom up process is followed for program design.
12. Example: Pascal, Fortran	12. Example: C++, Java.

FEATURES OF OOP

· **Class:** In object-oriented programming, a class is a programming language construct that is used as a blueprint to create objects. This blueprint includes attributes and methods that the created objects all share. Usually, a class represents a person, place, or thing - it is an abstraction of a concept within a computer program. Fundamentally, it encapsulates the state and behavior of that which it conceptually represents. It encapsulates state through data placeholders called member variables; it encapsulates behavior through reusable code called methods.

· **Object:** An Object is a real time entity. An object is an instance of a class. Instance means physically happening. An object will have some properties and it can perform some actions. Object contains variables and methods. The objects which exhibit similar properties and actions are grouped under one class. "To give a real world analogy, a house is constructed according to a specification. Here, the specification is a blueprint that represents a class, and the constructed house represents the object".

- To access the properties and methods of a class, we must declare a variable of that class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.
- We must acquire an actual, physical copy of the object and assign it to that variable. We can do this using **new** operator. The new operator dynamically allocates memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

· **Encapsulation:** Wrapping up of data (variables) and methods into single unit is called Encapsulation. Class is an example for encapsulation. Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class. Encapsulation is the technique of making the fields in a class private and providing access to the fields via methods. If a field is declared private, it cannot be accessed by anyone outside the class.

· **Abstraction:** Providing the essential features without its inner details is called abstraction (or) hiding internal implementation is called Abstraction. We can enhance the internal implementation without effecting outside world. Abstraction provides security. A class contains lot of data and the user does not need the entire data. The advantage of abstraction is that every user will get his own view of the data according to his requirements and will not get confused with unnecessary data. A bank clerk should see the customer details like account number, name and balance amount in the account. He should not be entitled to see the sensitive data like the staff salaries, profit or loss of the bank etc. So such data can be abstracted from the clerks view.

· **Inheritance:** Acquiring the properties from one class to another class is called inheritance (or) producing new class from already existing class is called inheritance. Reusability of code is main advantage of inheritance. In Java inheritance is achieved by using extends keyword. The properties with access specifier private cannot be inherited.

· **Polymorphism:** The word polymorphism came from two Greek words 'poly' means 'many' and 'morphos' means 'forms'. Thus, polymorphism represents the ability to assume several different forms. The ability to define more than one function with the same name is called Polymorphism

e.g.: int add (int a, int b)
float add (float a, int b)
float add (int a , float b)
void add (float a)
int add (int a)

· **Message Passing:** Calling a method in a class is called message passing. We can call methods of a class by using object with dot operator as:

object_name.method_name ();
e.g.: s.display (); ob.add (2, 5); ob.printMyName ();

INTRODUCING CLASSES

- ✓ A class is that it defines a new data type.
- ✓ Once defined, this new type can be used to create objects of that type.
- ✓ Thus, a class is a *template* for an object, and an object is an *instance* of a class

The General Form of a Class

A class is declared by use of the **class** keyword. The general form of a **class** definition is shown here:

```
Access specifier class classname {
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list) {
// body of method
}
type methodname2(parameter-list) {
// body of method

}
// ...
type methodnameN(parameter-list) {
// body of method
}
}
```

The data, or variables, defined within a **class** are called ***instance variables***. The code is contained within ***methods***. Collectively, the methods and variables defined within a class are called ***members*** of the class.

Eg:
class Box {
double width;
double height;
double depth;
}

```
class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;
// assign values to mybox's instance variables
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
// compute volume of box
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
}
}
```

Declaring Objects

Obtaining objects of a class is a two-step process.

First, declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object.
Second, acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator. The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

The above two statements can be written as a single statement as

```
Box mybox = new Box();
```

Introducing Methods

This is the general form of a method:

```
modifier type name(parameter-list)
{
// body of method
}
```

More generally, method declarations have six components, in order:

1. Modifiers—such as `public`, `private`, and others you will learn about later.
2. The return type—the data type of the value returned by the method, or `void` if the method does not return a value.
3. The method name—the rules for field names apply to method names as well, but the convention is a little different.
4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, `()`. If there are no parameters, you must use empty parentheses.
5. An exception list—to be discussed later.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

INTRODUCING ACCESS CONTROL

Encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: *access control.*
Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse.

Java's access specifiers are:

- **default**
- **public**
- **private**
- **protected**

- ✓ When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code.
- ✓ When a member of a class is specified as **private**, then that member can only be accessed by other members of its class.
- ✓ When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package. In the classes developed so far, all members of a class have used the default access mode, which is essentially public.

Here is an example:

```
class Test {
int a; // default access
public int b; // public access
private int c; // private access
// methods to access c
```

```
void setc(int i) { // set c's value
c = i;
}
int getc() { // get c's value
return c;
}
}
class AccessTest {
public static void main(String args[]) {
Test ob = new Test();
// These are OK, a and b may be accessed directly
ob.a = 10;
ob.b = 20;
// This is not OK and will cause an error
// ob.c = 100; // Error!
// You must access c through its methods
ob.setc(100); // OK
System.out.println("a, b, and c: " + ob.a + " " +
ob.b + " " + ob.getc());
}
}
```

CONSTRUCTORS

- ✓ A *constructor* initializes an object immediately upon creation.
- ✓ It has the same name as the class in which it resides and is syntactically similar to a method.
- ✓ Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes.
- ✓ Constructors have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself.

```
class Box {
double width;
double height;
double depth;

Box() {
System.out.println("Constructing Box");
width = 10;
height = 10;
depth = 10;
}
```

```
double volume() {
return width * height * depth;
}
}

class BoxDemo {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;

vol = mybox1.volume();
System.out.println("Volume is " + vol);

vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

Parameterized Constructors

The constructors that take parameters are called **parameterized constructors**.

```
class Box {
double width;
double height;
double depth;

Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}

double volume() {
return width * height * depth;
}
}

class BoxDemo {
public static void main(String args[]) {
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
```

```
vol = mybox1.volume();
System.out.println("Volume is " + vol);

vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

The output from this program is shown here:
Volume is 3000.0
Volume is 162.0

The this Keyword

- ✓ Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword.
- ✓ **this** can be used inside any method to refer to the *current o*bject.
- ✓ That is, **this** is always a reference to the object on which the method was invoked.
- ✓ You can use **this** anywhere a reference to an object of the current class' type is permitted.

```
Box(double w, double h, double d) {
this.width = w;
this.height = h;
this.depth = d;
}
```

Instance Variable Hiding:

- ✓ As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes.
- ✓ We can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables.
- ✓ However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable.
- ✓ This is why **width**, **height**, and **depth** were not used as the names of the parameters to the **Box( )** constructor inside the **Box** class. If they had been, then **width** would have referred to the formal parameter, hiding the instance variable **width**. While it is usually easier to simply use different names, there is another way around this situation.
- ✓ Because **this** lets you refer directly to the object, you can use it **to resolve any name space collisions** that might occur between instance variables and local variables.

```
// Use this to resolve name-space collisions.
```

```
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;    }
```

GARBAGE COLLECTION

- ✓ Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation.
- ✓ In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator.
- ✓ Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection.*
- ✓ It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program.

The finalize( ) Method

- ✓ Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed.
- ✓ To handle such situations, Java provides a mechanism called *finalization.*
- ✓ By using finalization, we can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
- ✓ To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class.
- ✓ Inside the **finalize( )** method you will specify those actions that must be performed before an object is destroyed.
- ✓ The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects.
- ✓ Right before an asset is freed, the Java run time calls the **finalize( )** method on the object.

The **finalize( )** method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

OVERLOADING METHODS

- ✓ In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different.
- ✓ When this is the case, the methods are said to be *overloaded*, and the process is referred to as *method overloading.*
- ✓ Method overloading is one of the ways that Java implements polymorphism.
- ✓ When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
- ✓ Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.
- ✓ When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Here is a simple example that illustrates method overloading:

```
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return a*a;
}
}

class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
```

```
System.out.println("Result of ob.test(123.25): " + result);
}
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

ARGUMENT PASSING

In general, there are two ways that a computer language can pass an argument to a subroutine.

1. Call-by-value
2. Call-by-reference

- ✓ The first way is *call-by-value*. This method copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.
- ✓ The second way an argument can be passed is *call-by-reference*. In this method, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

As you will see, Java uses both approaches, depending upon what is passed. In Java, when you pass a simple type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method.

```
// Simple types are passed by value.
class Test {
void meth(int i, int j) {
i *= 2;
j /= 2;
}
}
class CallByValue {
public static void main(String args[]) {
Test ob = new Test();
int a = 15, b = 20;
System.out.println("a and b before call: " +
a + " " + b);
ob.meth(a, b);
System.out.println("a and b after call: " +
a + " " + b);
```

```
}
}
```

The output from this program is shown here:
a and b before call: 15 20
a and b after call: 15 20

```
// Objects are passed by reference.
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// pass an object
void meth(Test o) {
o.a *= 2;
o.b /= 2;
}
}
class CallByRef {
public static void main(String args[]) {
Test ob = new Test(15, 20);
System.out.println("ob.a and ob.b before call: " +
ob.a + " " + ob.b);
ob.meth(ob);
System.out.println("ob.a and ob.b after call: " +
ob.a + " " + ob.b);
}
}
```

This program generates the following output:
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10

RECURSION

- ✓ Java supports *recursion.*
- ✓ Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself.
- ✓ A method that calls itself is said to be *recursive.*
- ✓ The classic example of recursion is the computation of the factorial of a number. The factorial of a number *N* is the product of all the whole numbers between 1

and *N*. For example, 3 factorial is 1 × 2 × 3, or 6. Here is how a factorial can be computed by use of a recursive method:

```
// A simple example of recursion.
class Factorial {
// this is a recursive function
int fact(int n) {
int result;
if(n==1) return 1;
result = fact(n-1) * n;
return result;
}
}

class Recursion {
public static void main(String args[]) {
Factorial f = new Factorial();
System.out.println("Factorial of 3 is " + f.fact(3));
System.out.println("Factorial of 4 is " + f.fact(4));
System.out.println("Factorial of 5 is " + f.fact(5));
}
}
```

The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

- ✓ When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start.
- ✓ A recursive call does not make a new copy of the method. Only the arguments are new.
- ✓ As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method.
- ✓ Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional function calls.

Disadvantage:

Many recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted. If this occurs, the Java run-time system will cause an exception.

Advantage:
The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives.

INTRODUCING NESTED AND INNER CLASSES

- ✓ It is possible to define a class within another class; such classes are known as *nested classes*.
- ✓ The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B is known to A, but not outside of A.
- ✓ A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.

- ✓ There are two types of nested classes: *static* and *non-static*.
- ✓ A static nested class is one which has the **static** modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.
- ✓ The most important type of nested class is the *inner* class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do. An inner class is fully within the scope of its enclosing class.

```
// Demonstrate an inner class.
class Outer {
int outer_x = 100;
void test() {
Inner inner = new Inner();
inner.display();
}
// this is an inner class
class Inner {
void display() {
System.out.println("display: outer_x = " + outer_x);
}
}
}
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();
}
}
```

Output from this application is shown here: display: outer_x = 100

```
// This program will not compile.
class Outer {
int outer_x = 100;
void test() {
Inner inner = new Inner();
inner.display();
}
// this is an inner class
class Inner {
int y = 10; // y is local to Inner
void display() {
System.out.println("display: outer_x = " + outer_x);
}
}
void showy() {
System.out.println(y); // error, y not known here!
}
}
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();
}
}
```

EXPLORING THE STRING CLASS

String is probably the most commonly used class in Java's class library.
Every string you create is actually an object of type **String**. Even string constants are actually **String** objects.
For example, in the statement System.out.println("This is a String, too"); the string "This is a String, too" is a **String** constant.
The second thing to understand about strings is that objects of type **String** are immutable; once a **String** object is created, its contents cannot be altered.
■ If you need to change a string, you can always create a new one that contains the modifications.
■ Java defines a peer class of **String**, called **StringBuffer**, which allows strings to be altered, so all of the normal string manipulations are still available in Java.

Strings can be constructed a variety of ways. The easiest is to use a statement like this:

```
String myString = "this is a test";
```

Once you have created a **String** object, you can use it anywhere that a string is allowed.

Java defines one operator for **String** objects: **+**. It is used to concatenate two strings.

For example, this statement
String myString = "I" + " like " + "Java.";
results in **myString** containing "I like Java."

```
// Demonstrating Strings.
class StringDemo {
public static void main(String args[]) {
String strOb1 = "First String";
String strOb2 = "Second String";
String strOb3 = strOb1 + " and " + strOb2;
System.out.println(strOb1);
System.out.println(strOb2);
System.out.println(strOb3);
}
}
```

The output produced by this program is shown here:
First String
Second String
First String and Second String

test two strings for equality by using **equals( )**.
obtain the length of a string by calling the **length( )** method.
obtain the character at a specified index within a string by calling **charAt( )**.

The general forms of these three methods are shown here:

boolean equals(String *object*)
int length( )
char charAt(int *index*)

Here is a program that demonstrates these methods:

```
// Demonstrating some String methods.
class StringDemo2 {
public static void main(String args[]) {
String strOb1 = "First String";
String strOb2 = "Second String";
String strOb3 = strOb1;
System.out.println("Length of strOb1: " +
strOb1.length());
System.out.println("Char at index 3 in strOb1: " +
strOb1.charAt(3));
if(strOb1.equals(strOb2))
```

```
System.out.println("strOb1 == strOb2");
else
System.out.println("strOb1 != strOb2");
if(strOb1.equals(strOb3))
System.out.println("strOb1 == strOb3");
else
System.out.println("strOb1 != strOb3");
}
}
```

This program generates the following output:

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```