Fifth Semester B.E. Degree Examination, Dec. 2019/Jan.2020
Programming in Java (15CS561)

Module-1

1 a. Explain three OOP principles (07 Marks)

The three OOP principles are

1. Encapsulation
2. Inheritance
3. Polymorphism

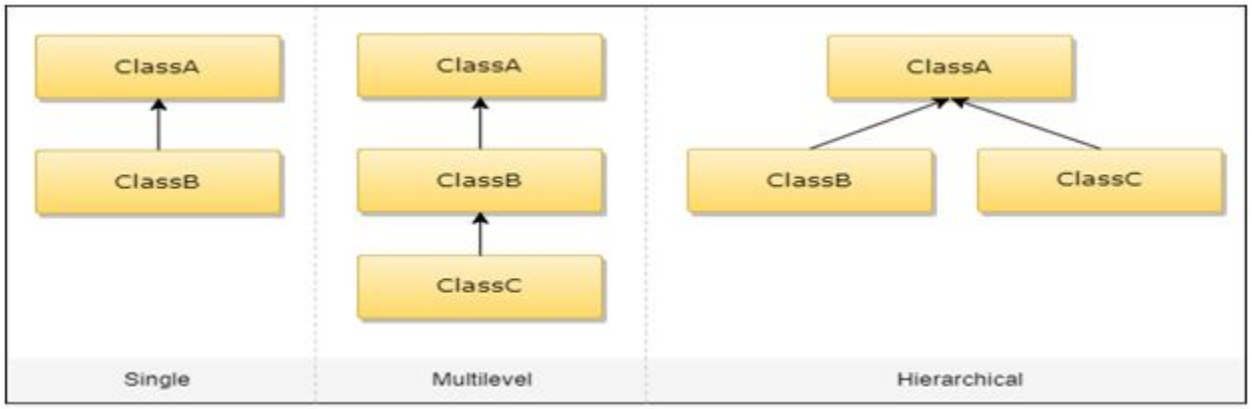
1. Encapsulation: Encapsulation can be defined as the procedure of casing up of codes and their associated data jointly into one single component.

In simple terms, encapsulation is a way of packaging data and methods together into one unit. Encapsulation gives us the ability to make variables of a class keep hidden from all other classes of that program or namespace.

Hence, this concept provides programmers to achieve data hiding. Programmers can have full control over what data storage and manipulation within the class

Inheritance: Inheritance can be defined as the procedure or mechanism of acquiring all the properties and behavior of one class to another, i.e., acquiring the properties and behavior of child class from the parent class.

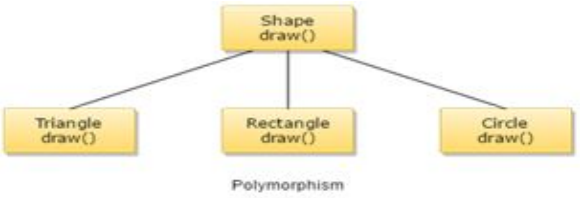
Java supports three types of inheritance. These are:



Java Inheritance

Polymorphism: The word polymorphism means having multiple forms. The term Polymorphism gets derived from the Greek word where poly + morphos where poly means many and morphos means forms.

- Static Polymorphism
- Dynamic Polymorphism.



b. Briefly explain any six features of Java. (06 Marks)

Six features of JAVA are

1. Simple
2. Secure
3. Portable
4. Object-Oriented
5. Robust
6. Multithreaded

1. Simple:

Java was designed to be easy for the professional programmer. For those who have already understood the basic concepts of object-oriented programming, and for an experienced C++ programmer learning Java will be even easier as **Java inherits the C/C++ syntax and many of the object-oriented features of C++.**

2. Secure

Java provides security. **The security is achieved by confining an applet to the Java execution environment and not allowing it access to other parts of the computer.** The ability to download applets with confidence that no harm will be done and that no security will be breached is considered by many to be the single most innovative aspect of Java.

3. Portable:

Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there need to be some way to enable the program to execute on different systems. **Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform.** Once the run-time package exists for a given system, any Java program can run on it.

4. Object-Oriented:

Java has a clean, usable, pragmatic approach to objects. The object model in Java is simple and easy to extend. The primitive types, such as integers, are kept as high-performance non-objects.

5. Robust:

The ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts programmer in a few key areas to force the programmer to find mistakes early in program development. **Also, Java frees the programmer from having to worry about many of the most common causes of programming errors.** As **Java is strictly typed language, it checks code not only at run time but also during compilation time. As a result, many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java.**

The two features – Garbage collection and Exception handling enhance the robustness of Java Programs.

a) Garbage Collection:
In C/C++, the programmer must manually allocate and free all dynamic memory which sometimes 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 their code is still using. **Java eliminates these problems by managing memory allocation and de-allocation. De-allocation is completely automatic because Java provides garbage collection for unused objects.**

b) Exception Handling:
Exceptional conditions in traditional environment arise in situations such as "**division by zero" or "file not found"** which are managed by clumsy and hard-to-read constructs. Java helps in this area by providing **object oriented exception handling.**

6. Multi threaded
Java supports multithreaded programming, which allows the programmer to write programs that do many things simultaneously. Java provides an elegant solution for multi process synchronization that enables the programmer to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows the programmer to think about the specific behaviour of the program rather than the multitasking subsystem.

c. Define type casting. Explain two types of conversion. (07 Marks)

Type Casting:

- If we want to **assign an int value to a byte variable**, conversion will not be performed automatically, because a byte is smaller than an int.
- This kind of conversion is called **narrowing conversion** since we are explicitly making the value narrower so that it will fit into the target type.
- **To create a conversion between the two incompatible types, we must use a cast. A cast is simply an explicit type conversion. Type casting is the process of conversion between two incompatible types.**
- The general form of type cast is -

(target-type) value

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

For example to cast an int to a byte
int a=20;
byte b;
b= (byte) a;

- If the **integer value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.**
- A different type of conversion called **truncation** will occur when a floating-point value is assigned to an integer type. Integers do not have fractional components. Hence **when a floating point value is assigned to an integer type, the fractional component is lost.**
- For example, if the value 1.23 is assigned to an integer, the resulting value will be 1. The 0.23 will be truncated.
- **If the size of the whole number component is too large to fit into the target integer type, then the value will be reduced modulo the target type's range.**

The two types of conversion are

1. Widening conversion

2. Narrowing conversion

1. Widening Conversion:

- 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 widening conversions, the numeric types, including integer and floating-point types are compatible with each other.
- There is **no automatic conversions from the numeric types to char or boolean**. Also char or boolean are not compatible with each other.
- Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long or char.

2. Narrowing Conversion:

- If we want to **assign an int value to a byte variable**, conversion will not be performed automatically, because a byte is smaller than an int.
- This kind of conversion is called **narrowing conversion** since we are explicitly making the value narrower so that it will fit into the target type.
- **To create a conversion between the two incompatible types, we must use a cast. A cast is simply an explicit type conversion.**

- The general form of cast is -

(target-type) value

target-type specifies the desired type to convert the specified value to.
For example to cast an int to a byte

int a=20;
byte b;
b= (byte) a;

- f the **integer value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.**
- A different type of conversion called **truncation** will occur when a floating-point value is assigned to an integer type. Integers do not have fractional components. Hence **when a floating point value is assigned to an integer type, the fractional component is lost.**
- For example, if the value 1.23 is assigned to an integer, the resulting value will be 1. The 0.23 will be truncated.
- **If the size of the whole number component is too large to fit into the target integer type, then the value will be reduced modulo the target type's range.**

2 a. Expain how arrays are defined in JAVA with an example. (07 Marks)

One-Dimensional Arrays:

Syntax for defining a one-dimensional array:
type array-var [ ] = new type [size];

Example:
int month_days[] = new int [12];

```
// Program to demonstrate one-dimensional array
class Array
{
   public static void main(String args[])
   {
      int 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. ");
    }
}
```

MULTI DIMENSIONAL ARRAYS:

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

TWO DIMENSIONAL ARRAYS:

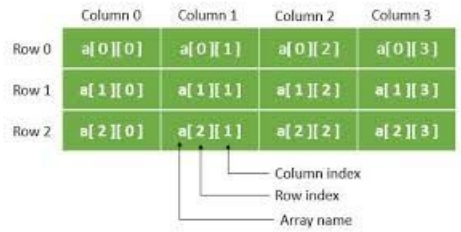
Syntax:

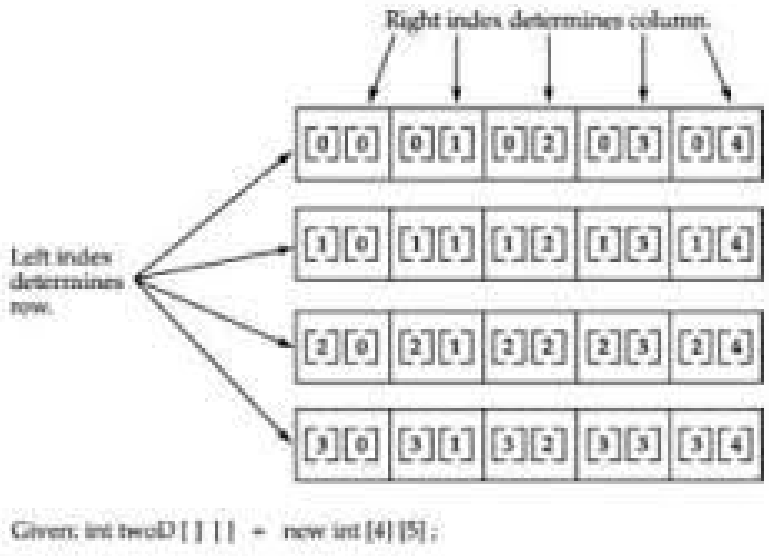
type var-name[ ] [ ];
var-name = new type [size] [size];
or
type var-name [ ] [ ] = new type [size] [size];

Example

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

The figure shows the conceptual view of a 3 by 4 , two dimensional array





The following program demonstrates 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();
      }
   }
}
```

UNEVEN OR IRREGULAR MULTIDIMENSIONAL ARRAY:

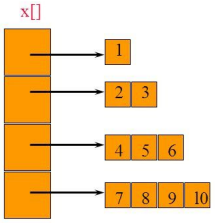
- When you allocate memory for a multidimensional array, we need to only specify the memory for the first (leftmost) dimension. Then we can allocate the remaining dimensions separately.
- Example

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

```
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

- The advantage of allocating the second dimension array separately is we need not allocate the same number of elements for each dimension.

Array Of Arrays Representation



Use an irregular 2-D array … length of rows is not required to be the same.

```
[0][0]
   [1][0] [1][1]
   [2][0] [2][1] [2][2]
   [3][0] [3][1] [3][2] [3][3]
```


The following program creates a two dimensional array in which the size of the second dimension is unequal.

```
// program to demonstrate differing size second dimension
class TwoDAgain
{
   public static void main(String args[])
   {
      int twoD[][] = new int [4][];
      twoD[0] = new int[1];
      twoD[1] = new int[2];
      twoD[2] = new int[3];
      twoD[3] = new int[4];
      int i, j, k=0;
      for(i=0;i<4;i++)
         for(j=0;j<i+1;j++)
```

```
            {
               twoD[i][j] = k;
               k++;
            }
        for(i=0;i<4;i++)
        {
            for(j=0;j<i+1;j++)
              System.out.print(twoD[i][j] + " ");
             System.out.println();
        }
    }
}
```

The above program generates the following output:

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

b. Explain the process of compiling and running Java application with help of "Hello world" program. (06 Marks)

Simple Java program -

```
/*This is a simple Java program.
Call this file "Example.java".
*/
class Example
{
        // Your program begins with a call to main().
        public static void main(String args[])
        {
                System.out.println("Hello world");
        }
}
```

The following are the steps involved in compiling and running a Java program

a) After typing the program, in the terminal we have to type **javac program_name.java** and hit enter
Example:
javac Example.java

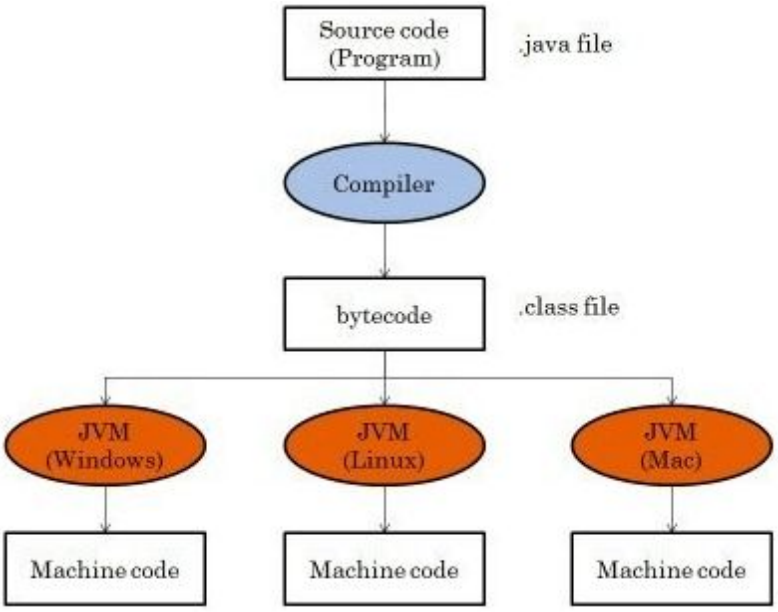
The javac is the java compiler. It converts the java source code to byte code. Byte code have the extension .class. As per this example, the file Example.class will be created.

b) Then again in the terminal (if no error is there) then we have to type java program name and hit the enter key
Example:
java Example

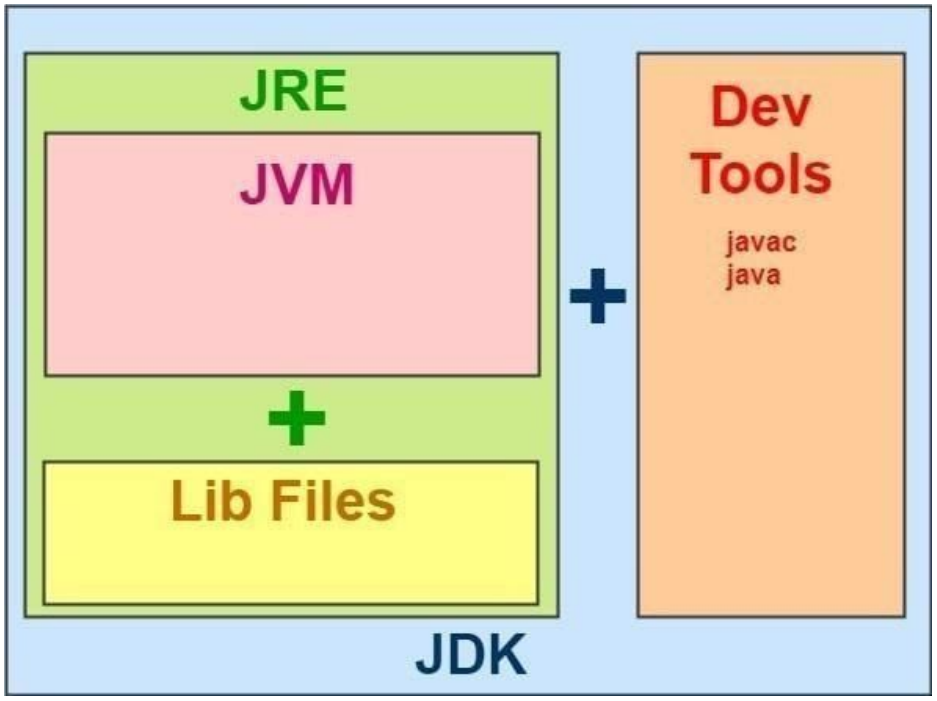
java is the interpreter which interpretes the byte code, Example.class. The output obtained is
Hello world

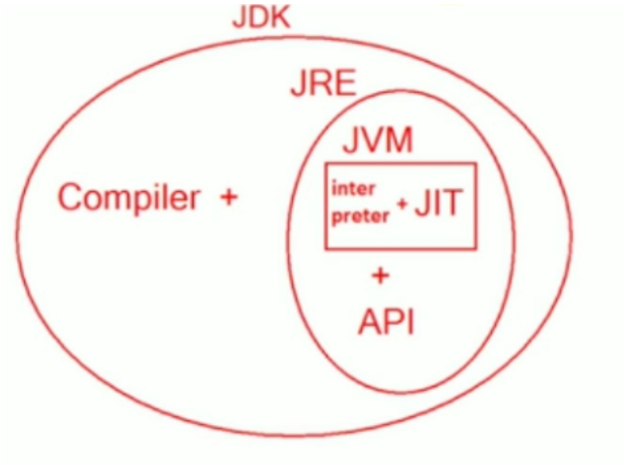
c. Define byte code How does it help in Java program to achieve portability? (07 Marks)

- **Bytecode** is the highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM).
- JVM is designed as an interpreter for bytecode.
- Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform.
- Once the JVM exists for a given system, any Java program can run on it. Note that the details of the JVM will differ from platform to platform, but all understand the same Java bytecode.
- If the Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is not the solution, thus the execution of bytecode by the JVM is the easiest way to create truly **portable** programs.



The following two figures show how JDK fits into the Java application development lifecycle.





JDK = JRE + Development/debugging tools
JRE = JVM + Java Packages Classes (like util, math, lang, awt,swing etc)+runtime libraries.
JVM = Interpreter + JIT

- **The JVM is the Java platform component that executes programs.** It is responsible for running Java bytecodes. It provides interpreter plus JIT for running bytecode by converting into current OS machine language.
- The JRE is the **on-disk** part of Java that creates the JVM. It provides only run time environment and does not provide any development tools. Hence by using JRE software, we can only execute already developed applications. We cannot develop new applications and also we cannot modify existing applications. **The JRE can be used as a standalone component to simply run Java programs, but it is also a part of JDK. The JDK requires a JRE because running Java programs is part of developing them. JRE** provides the libraries, the Java Virtual Machine (JVM) and other components to run applets and applications written in the Java programming language.
- **JDK** is a superset of the JRE, and contains everything that is in **JRE**, plus tools such as compilers and debuggers necessary for developing applets and applications. The JDK allows developers to create Java programs that can be executed and run by the JVM and JRE. Using JDK we can develop, compile and execute new applications and modify existing applications.
- **The distinction between JDK and JRE is that the JDK is a package of tools for *developing* Java-based software, whereas the JRE is a set of tools for *running* Java code.**

Module-2

3 a. Explain with example (i) Labelled continue (ii) >>> (iii) Short circuit AND operator. (06 Marks)

(i) Labelled continue:

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:

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

(ii) >>>

>>> - The Unsigned Right Shift:

To preserve the sign of the value, the >> operator fills the high order bit with its previous contents each time a shift occurs. This is undesirable, if we are shifting a non-numeric value (something that does not represent a numeric value). This situation arises when working with graphics.

We want to shift a zero into the high order bit irrespective of its initial value. This is known as **unsigned shift**. For this, we use Java's **unsigned shift – right operator, >>>, which always shifts zero into the high order bit.**

The following code demonstrates >>>
int a = -1;
a = a >>> 24

1111 1111 1111 1111 1111 1111 1111 1111 -1 in binary as an int
>>> 24
0000 0000 0000 0000 0000 0000 1111 1111 255 in binary as an int

The >>> operator is meaningful for 32-bit and 64-bit values. As smaller values are automatically promoted to int in expressions.
For unsigned right shift on a byte value zero filling must begin at bit 7. But this is not the case, since it is a 32-bit value that is actually being shifted.

(iii) Short cirtuit AND operator

&& - Short-Circuit Logical AND Operator:

- The AND operator results in false when A is false, no matter what B is.
- **When we use && form, rather than & form of this operator, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone.**
- Example
  if (denom !=0 && num /denom > 10)
  As the short circuit form of && 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.
- Its a standard practice to use the short-circuit forms of AND and OR in cases involving Boolean logic, leaving the single character versions exclusively for bitwise operations.

b. Explain bitwise shift and bitwise logical operators. (06 Marks)

The bitwise shift operators are

1. The left shift operator
2. The right shift operator
3. The unsigned right shift operator

1. THE LEFT SHIFT:

- The left shift operator, <<, shifts all of the bits in a value to the left a specified number of times.
- The general form is
  value << num
  num specifies the number of positions to left-shift the value in value.
- For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right.
- When a left shift is applied to an int operand, bits are lost once they are shifted past bit position 31.
- When a left shift is applied to a long operand, bits are lost once they are shifter past bit position 63.
- Java's automatic type promotions produce unexpected results when we are shifting byte and short values.
- For example, if we left shift a byte value, that value will first be promoted to int and then shifted. Therefore, we must discard the top three bytes of the result if we need the shifted byte value. For this, we need to cast the result back to byte.
- The following program demonstrates this concept

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

Output:

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

a = 64 => 0000 0000 0000 0000 0000 0000 0100 0000
i = a << 2 = 256 => 0000 0000 0000 0000 0000 0001 **0000 0000**
b = (byte) (a << 2) = 0 => **0000 0000**

- Each left shift doubles the original value. But if we shift a 1 into the high-order position (bit 31 or 63) the value will become negative.
- Example program to show left shift multiplies the value by 2

```
// class MultByTwo {
  public static void main(String args[ ] ) {
    int i;
```

```
        int num = 0xFFFF FFE;
        for (i = 0; i < 4 ; i++) {
            num = num << 1;
            System.out.println(num);
        }
    }
}
```

Output:

536870908
1073741816
2147483632
-32

num=0xFFFF FFE => 0000 1111 1111 1111 1111 1111 1111 1110
i=0; num = num << 1 = 536870908 => 0001 1111 1111 1111 1111 1111 1111 1100
i=1; num = num << 1 = 1073741816 => 0011 1111 1111 1111 1111 1111 1111 1000
i=2; num = num << 1 = 2147483632 => 0111 1111 1111 1111 1111 1111 1111 0000
i=3; num = num << 1 = -32 => 1111 1111 1111 1111 1111 1111 1110 0000

2. The Right Shift:

- The right shift operator, >>, shifts all the bits in a value to the right a specified number of times.
- General form:
  value >> num;
  where num specifies the number of positions to right shift the value in value.
- Example:
  int a = 32;
  a = a >> 2; // a now contains 8

a = 32 => 0000 0000 0000 0000 0000 0000 0010 0000
a = a >> 2 = 8 => 0000 0000 0000 0000 0000 0000 0000 1000

- When the bits are shifted off (low order bits) those bits are lost.
- Example:
  int a = 35;
  a = a >> 2; // a still contains 8

a = 35 => 0000 0000 0000 0000 0000 0000 0010 0011
a = a >> 2 = 8 => 0000 0000 0000 0000 0000 0000 0000 1000

- Each time we shift a value to the right, it divides that value by 2 and discards any remainder.

- The >> operator automatically fills the high order bit with its previous contents each time a shift occurs. That is, when we are shifting right, the (leftmost) top bits fills the previous contents of the top bit. This is called **sign extension** and serves to preserve the sign of negative numbers.

- For example -8 >> 1 is -4

1111 1000 -8
>> 1
1111 1100 -4

- If we shift -1 right, the result always remains -1, as sign extension keeps bringing in more ones in the high order bits.

3. The Unsigned Right Shift:

- To preserve the sign of the value, the >> operator fills the high order bit with its previous contents each time a shift occurs. This is undesirable, if we are shifting a non-numeric value (something that does not represent a numeric value). This situation arises when working with graphics.
- We want to shift a zero into the high order bit irrespective of its initial value. This is known as **unsigned shift**. For this, we use Java's **unsigned shift – right operator, >>>, which always shifts zero into the high order bit.**
- The following code demonstrates >>>

```
int a = -1;
a = a >>> 24
```

1111 1111 1111 1111 1111 1111 1111 1111 -1 in binary as an int
>>> 24
0000 0000 0000 0000 0000 0000 1111 1111 255 in binary as an int

- The >>> operator is meaningful for 32-bit and 64-bit values. As smaller values are automatically promoted to int in expressions.
- For unsigned right shift on a byte value zero filling must begin at bit 7. But this is not the case, since it is a 32-bit value that is actually being shifted.

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
0	1	1	0	1	1

1	0	1	0	1	0
1	1	1	1	0	0

The Bitwise NOT:

- The bitwise NOT operator, ~, inverts all of the bits of its operand.
- It is also called bitwise complement or unary NOT operator.
- Its an unary operator.
- For example,

```
int x = 42;
int y;
y = ~x;
```

x = 42 0 0 1 0 1 0 1 0 1 0
~x 1 1 0 1 0 1 0 1 0 1 = -43
so, y = -43

The Bitwise AND:

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

```
int x = 42;
int y = 15;
int z = x & y;
```

```
  0 0 1 0 1 0 1 0      42
& 0 0 0 0 1 1 1 1      15
-----------------------------
  0 0 0 0 1 0 1 0      10
```

so z = 10

The Bitwise OR:

- The bitwise OR operator, |, produces 1 if either of the bits in the operands is a 1.
- Example:

```
int x = 42;
int y = 15;
int z = x | y;
```

```
  0 0 1 0 1 0 1 0      42
| 0 0 0 0 1 1 1 1      15
-----------------------------
  0 0 1 0 1 1 1 1      47
```

so z = 47

The Bitwise XOR:

- The bitwise XOR operator, ^, produces 1 if exactly one operand is 1 otherwise the result is zero.
- Example:

int x = 42;
int y = 15;
int z = x ^ y;

```
   0 0 1 0 1 0 1 0        42
^  0 0 0 0 1 1 1 1        15
-------------------------------
   0 0 1 0 0 1 0 1        37
```

so z = 37

c. With code snippets, explain jump statements in Java (08 Marks)

JUMP STATEMENTS:

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

1. break:

break statements has three uses.

- First it terminates a statement sequence in a switch statement.
- Second it can be used to exit a loop.
- Third, it can be used as goto.

Using a break to exit a loop:

- When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop.
- The break statement can be used with any of Java's loops, including intentionally infinite loops.
- 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:

- goto is useful when you are exiting from a deeply nested set of loops.
- Goto is not preferred as a goto-ridden code is hard to understand and hard to maintain. But they can be useful when you are exiting from a deeply nested set of loops.
- Break gives the benefits of a goto statement without its problems
- The general form

  break label;
- label is the name of a label that identifies a block of code.
- A label is any valid Java identifier followed by a colon.
- Example:

```
class Break
 {
```

```
  public static void main(String args[ ])
  {
     boolean t = true;
     first:
     {
        second :
        {
              third:
              {
                 System.out.println("Before the break");
                 if (t) break second; // break out of second block
                 System.out.println("This won't execute");
              }    // End of third block
              System.out.println("This won't execute");
        }
        System.out.println("This is after second block .");
     }   // End of first block
  }   // End of main
}
```

Output:

Before the break

This is after second block

USING CONTINUE:

- useful to force an early iteration of a loop.
- In while and do-while loops a continue statement causes control to be transferred directly to the conditional expression that controls the loop.
- In a for loop, control goes first to the the iteration portion of the for statement and then to the conditional expression.
- As with the break statement, continue may specify a label to describe which enclosing loop to continue.

Return:

- The return statement is used to explicitly return from a method.
- It causes program control to transfer back to the caller of the method.
- The return statement immediately terminates the method in which it is executed.

4 a. Explain syntax of for each loop. Write a program to search key element by using for each loop. (06 Marks)

For-each loop or Enhanced 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. Java adds the for-each capability by enhancing the for statement. The advantage of this approach is that no new keyword is required, and no preexisting code is broken. The for-each style of for is also referred to as the **enhanced for loop.**

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

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

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

With each iteration of the loop, the next element in the collection is retrieved and stored in itr-var. The loop repeats until all elements in the collection have been obtained. Because the iteration variable receives values from the collection, type must be the same as (or compatible with) the elements stored in the collection. Thus, when iterating over arrays, type must be compatible with the base type of the array.

```
// Search an array using for-each style for.
class Search {
        public static void main(String args[]) {
                int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
                int val = 5;
                boolean found = false;
                // use for-each style for to search nums for val
                for(int x : nums) {
                        if(x == val) {
                                found = true;
                                break;
                        }
                }
                if(found)
                        System.out.println("Value found!");
                else
                        System.out.println("Value nt found!");

        }
```

}

b. What are operators? Explain any 5 types in detail. (06 Marks)

Operators are symbols that perform special operations on one, two or three operands and then return a result.

In Java, operators are divided into four groups
1. Arithmetic
2. Bitwise
3. Relational
4. Logical

1. Arithmetic Operators:

- Arithmetic operators are used in mathematical expressions.
- The operands of the arithmetic operators must be of numeric type.
- It can't be used on boolean type.
- It can be used on char type as char type in Java is a subset of int.
- The various arithmetic operators are shown in the table below

Sl. No.	Operator	Result
1	+	Addition
2	-	Subtraction (Also unary minus)
3	*	Multiplication
4	/	Division
5	%	Modulus
6	++	Increment
7	+=	Addition Assignment
8	-=	Subtraction Assignment
9	*=	Multiplication Assignment
10	/=	Division Assignment
11	%=	Modulus Assignment
12	--	Decrement

The Basic Arithmetic Operators:

- The basic arithmetic operations are -

1. addition
2. subtraction
3. multiplication
4. division

- The minus operator also has a unary form that negates its single operand.
- When the division operator is applied to an integer type, there will be no fractional component attached to the result.
- The following program demonstrates the arithmetic operations -

```
// Program to demonstrate the basic arithmetic operators
class BasicMath {
   public static void main( String args[ ]) {
      System.out.println("Integer Arithmetic");
      int a = 1 + 1;
      int b = a * 3;
      int c = b / 4;
      int d = c – a;
      int e = -d;

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


      System.out.println("Floating Point Arithmetic");
      double da = 1 + 1;
      double db= da * 3;
      double dc = db / 4;
      double dd = dc - da;
      double de = -dd;


      System.out.println("da = " + da);
      System.out.println("db = " + db);
      System.out.println("dc = " + dc);
      System.out.println("dd = " + dd);
      System.out.println("de = " + de);
   }
}
```

Output:

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

e = 1
Floating point arithmetic
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5

The modulus operator:

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

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


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

Output:

x mod 10 = 2
y mod 10 = 2.25

Arithmetic Compound Assignment Operators:

- Compound assignment operators are special operators that are used to combine an arithmetic operation with an assignment operation.
- Statement like the following
  a = a + 4;
  can be rewritten as
  a += 4;
- The above statement uses the += compound assignment operator. Both statements perform the same action. They increase the value of a by 4.
- There are compound assignment operators for all arithmetic, binary operators.
- Any statement of the form
  var = var op expression;
  can be rewritten s
  var op= expression;
- **Advantages:**
  1. They save a bit of typing because they are "shorthand" for their equivalent long forms.

2. They are implemented more efficiently by the Java run-time system than their equivalent long forms.

- Hence professionally written Java programs use compound assignment operators.
- The following program illustrates several op= assignments in action

```
// Demo program to illustrate compound assignment operators
class OpEquals {
   public static void main(String args [ ]) {
      int a = 1;
      int b = 2;
      int c = 3;


      a += 5;
      b *= 4;
      c += a * b;
      c %= 6;
      System.out.println("a = " + a);
      System.out.println("b = " + b);
      System.out.println("c = " + c);
   }
}
```

Output:

a = 6
b = 8
c = 3

Increment and Decrement operator:

- The ++ and – are Java's increment and decrement operators respectively.
- The increment operator increases its operand by one.
- The decrement operator decreases its operand by one.
- The statement

  x = x + 1;

  can be rewritten in Java using increment operator as

  x++;
- Similarly, the statement

  x = x – 1; is same as

  x--;
- Increment and decrement operators appear both in postfix form and prefix form.
- In postfix form, the operator follows the operand
- In prefix form the operator precede the operand.
- For statements like

  x++;

--y;

there is no difference between prefix and postfix forms.

- The prefix and postfix forms matter a lot when the increment and/or decrement operators are part of a larger expression.
- In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression.
- In the postfix form, the previous value is obtained for use in the expression and then the operand is modified.
- Prefix example, Consider the statements -

  x = 42;

  y = ++x;

  Here, the increment occurs before x is assigned to y. So y = 43 and x = 43. Thus, the above line is equivalent to following two statements

  x = x + 1;

  y = x;
- Postfix example: Consider the statements -

  x = 42;

  y = x++;

  Here, the value of x is obtained before the increment operator is executed. So y = 42 and x = 43. Thus, the above line is equivalent to following two statement -

  y = x;

  x = x + 1;
- The following program demonstrates the increment operator

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

Output:

a = 2

b = 3

c = 4
d = 1

c. Explain loops in Java. (08 Marks)

- Loops in Java are **for, while and do-while**.
- These statements create loops. A loop repeatedly executes the same set of instructions until a termination condition is met.

while:

- The while loop repeats a statement or block while its controlling expression is true.
- **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 not needed if only a single statement is being repeated.
- **As the while loop evaluates its conditional expression at the top of the loop, the body will not execute even once if the condition is false to begin with.**
- The body of the while can be empty. As a null statement is syntactically valid in Java.

The do-while:

- **The do-while loop always executes its body at least once, because its condition expression is at the bottom of the loop.**
- General form

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

- Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. Condition must be a Boolean expression.
- The do-while loop is useful when you process a menu selection, because you will usually want the body of the menu loop to execute at least once.

The for loop:

- **Beginning with JDK 5, there are two forms of the for loop.**
- The first is the traditional form that has been in use since the original version of Java.
- The second is the new "for-each" form.
- **General form of traditional for statement-**

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

- If only one statement is being repeated, there is no need for the curly braces.
- The for loop operates as follows. When the loop first starts, the initialization portion of the loop is executed. This sets the value of the **loop control variable**, which acts as a counter that controls the loop. Next, condition is evaluated. This must be a Boolean expression. If this expression is true, then the body of the loop is executed. Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the condition expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the condition returns false.
- **Declaring loop control variables inside the for loop:**
  - If the variable that controls a for loop is only needed for the purpose of the loop and is not used elsewhere, it is possible to declare the variable inside the initialization portion of the for.
  - When we declare a variable inside a for loop, the scope of that variable is limited to the for loop. Outside the for loop, the variable will cease to exist.
- **Using the Comma:**
  - Java permits the user to include multiple variables in both the initialization and iteration portions of the for loop. Each variable is separated from the next by a comma.
  - Example:

```
int a, b;
for (a=1, b=4; a < b; a++, b--) {
  System.out.println(a + " ");
  System.out.println(b + " ");
}
```

- **for loop variations:**
  - **First for loop variation:** The condition expression of the for loop does not need to test the loop control variable against some target value.

    Example:

```
boolean done = false;
for(int i=1;!done;i++) {
   ...
   if (interrupted()) done = true;
```

```
}
```

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

- **Second for loop variation:** Either the initialization or the iteration expression or both may be absent

Example:

```
boolean done = false;
int i = 0;
for (; !done; ) {
   if ( i== 10) done = true;
   i++;
}
```

- Third for loop variation: Infinite loop – a loop that never terminates

```
for ( ; ; ) {
// ...
}
```

The for-each version of the for loop:

- Beginning with JDK 5, a second form of for – for-each loop got introduced.
- A for-each style loop is designed to cycle through a collection of objects, such as an array, from start to finish.
- Java adds the for-each capability by enhancing the for statement. The advantage of this approach is that no new keyword is required and no preexisting code is broken.
- It is also known as enhanced for loop.
-
- **General form**

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

- Here, **type** specifies the type and **itr-var** specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end. The collection being cycled through is specified by collection. With each iteration of the loop, the next element in the collection is retrieved and stored in itr-var. The loop repeats until all elements in the collection have been obtained.
- **Example:**

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

- Although, the for-each for loop iterates until all elements in an array have been examined, it is possible to terminate the loop early by using a break statement.
- The 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. That is, we can’t change the contents of the array by assigning the iteration variable a new value.
- Example program

```
        // The for-each loop is 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;
              }
           System.out.println();
           for (int x: nums)
              System.out.print(x + " ");
           System.out.println();
        }
}
```

Output:

1 2 3 4 5 6 7 8 9 10

1 2 3 4 5 6 7 8 9 10

Module-3

5 a. What is super? Explain use of super with example. (06 Marks)

super is the keyword using which the subclass refers to its immediate superclass. super() always refers to the superclass immediately above the calling class. This is true even in multilevel hierarchy. super() must always be the first statement executed inside a subclass constructor.

The keyword super has two uses

1. To invoke the superclass' constructor.
2. To access a member of the superclass that has been hidden by member of a subclass.

1. Using super to call superclass constructors:

A subclass can call a constructor defined by its superclass by use of the following form of superclass

super(arg-list);

Here, arg-list specifies any arguments needed by the constructor in the superclass. super() must always be the first statement executed inside a subclass' constructor.

2. Second use of super to access a member of the superclass:

The second form of super always refers to the superclass of the subclass in which it is used.

The general form is

super.member

Here, member can be either a method or an instance variable. The second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

Example program to demonstrate both uses of super

```
// create a superclass
class A
{
   int    i, j ;
   A (int a, int b)
          {
      i = a;
      j = b;
   }
   void show()
          {
      System.out.println(" i =  " + i);
      System.out.println("j = " + j);
   }
}


// Define subclass
class B extends A
{
   int k;
   B(int a, int b, int c)
          {
      super(a,b);  // use 1 of superclass
      k = c;
   }
         void show()
         {
       super.show();     // calls show() of class a, use 2 of
superclass
      System.out.ptinln("k = " + k);
   }
}


class Demo
{
   public static void main(String args[])
          {
      B obj = new B(1,2,3);
      obj.show();
   }
```

}
Output: $javac Demo.java $java Demo i = 1 j = 2 k = 3

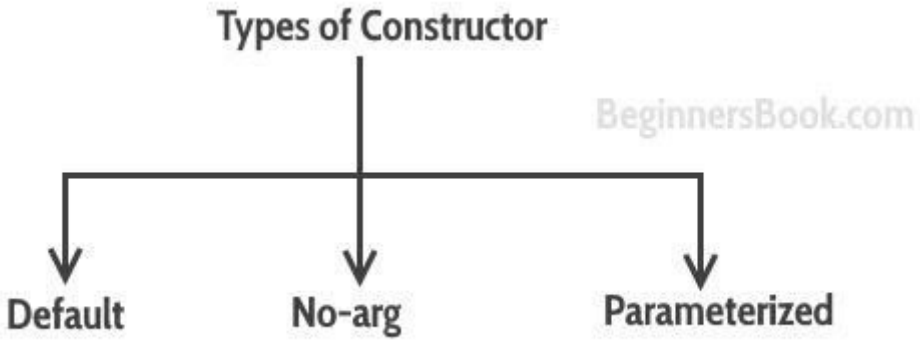
b. What is constructor? Explain its types with example. (07 Marks)

Constructor is a special type of member method which is invoked automatically when the object gets created. Constructors are used for object initialization. They have the same name as that of the class. Constructors are automatically called immediately after the object is created. Since they are called automatically, there is no return type, not even void. Constructors may or may not take parameters.

- Every class is provided with a ***default constructor*** which initializes all the data members to respective *default values*. (Default for numeric types is zero, for character and strings it is null and default value for Boolean type is false.)
- In the statement *classname ob= new classname();* the term *classname()* is actually a constructor call.
- If the programmer does not provide any constructor of his own, then the above statement will call default constructor.
- If the programmer defines any constructor, then default constructor of Java cannot be used.
- So, if the programmer defines any parameterized constructor and later would like to create an object without explicit initialization, he has to provide the default constructor by his own. For example, in the below program, if we remove ordinary constructor, the statements like Box b1=new Box(); will generate error. To avoid the error, we should write a default constructor like – Box(){ } Now, all the data members will be set to their respective default values.

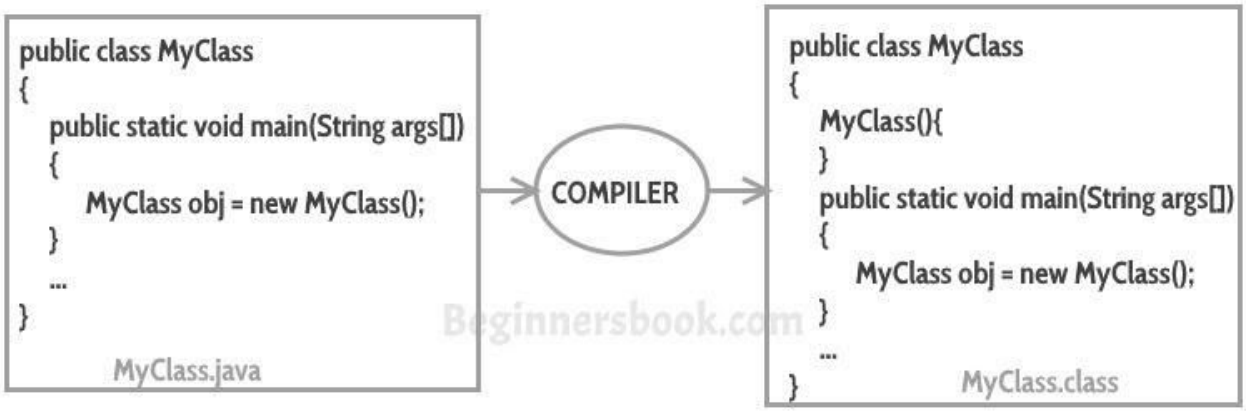
Different types of constructor are

1. Default constructor
2. Non parameterized or No-arg or zero argument constructor
3. Parameterized constructor



Default constructor

If you do not implement any constructor in your class, Java compiler inserts a default constructor into your code on your behalf. This constructor is known as default constructor. You would not find it in your source code(the java file) as it would be inserted into the code during compilation and exists in .class file. This process is shown in the diagram below:



If you implement any constructor then you no longer receive a default constructor from Java compiler.

No-arg constructor:

Constructor with no arguments is known as **no-arg constructor**. The signature is same as default constructor, however body can have any code unlike default constructor where the body of the constructor is empty.

Although you may see some people claim that that default and no-arg constructor is same but in fact they are not, even if you write **public Demo() { }** in your class Demo it cannot be called default constructor since you have written the code of it.

Parameterized constructor

Constructor with arguments(or you can say parameters) is known as Parameterized constructor

```
// program to describe the types of constructor.
class Box
{
   double w, h, d;

   Box() //non parameterized constructor
   {
     w=h=d=5;
   }

   Box(double wd, double ht, double dp) //parameterized constructor
   {
     w=wd;
     h=ht;
     d=dp;
   }
   double volume()
   {
     return w*h*d;
   }

}


class BoxDemo
{
   public static void main(String args[])
   {
     Box mybox1 = new Box( );
     Box mybox2 = new Box(2, 2, 2);

     double vol;
     // get volume of first box
     vol = mybox1.volume();
     System.out.println("Volume is " + vol);
     // get volume of second box
     vol = mybox2.volume();
```

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

Output:

```
$ javac BoxDemo.java
$ java BoxDemo
Volume is 125.0
Volume is 8.0
```

c. List and explain access specifiers in JAVA and their scope (07 Marks)

The access specifiers in JAVA are

1. private
2. public
3. protected

The three access specifiers, **private, public, and protected**, provide a variety of ways to produce the many levels of access required by these categories.

Public:

Anything declared **public** can be accessed from anywhere.

Private:

Anything declared **private** cannot be seen outside of its class.

Default:

When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the **default** access.

Protected:

If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.

A non-nested class has only two possible access levels: default and public. When a class is declared as public, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.

The table shows class member access

TABLE 9-1 Class Member Access

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

6 a. With example, explain finalize() method. (06 Marks)

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 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, you 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
}
```

Here, the keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class. It is important to understand that finalize( ) is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—finalize( ) will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on finalize( ) for normal program operation.

b. What is inheritance? Explain types of inheritance. (07 Marks)

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it.

A class that is inherited is called a **superclass**. The class that does the inheriting is called a **subclass**. Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

To inherit a class the keyword **extends** is used. Example program creates a superclass called A and a subclass called B.

```
// A simple example of inheritance.
// Create a superclass.
class A
{
   void showA()
   {
     System.out.println("show of parent class A");
   }
}


// Create a subclass by extending class A.
class B extends A
{
   void showB()
   {
     System.out.println("Show of child class B");
   }
}


class SimpleInheritance
{
   public static void main(String args[])
   {
     A superOb = new A();
     B subOb = new B();
     superOb.showA();
      /* The subclass has access to all public members
of
     its superclass. */
```

```
        subOb.showB();
        subOb.showA();

    }
}
```

Output:

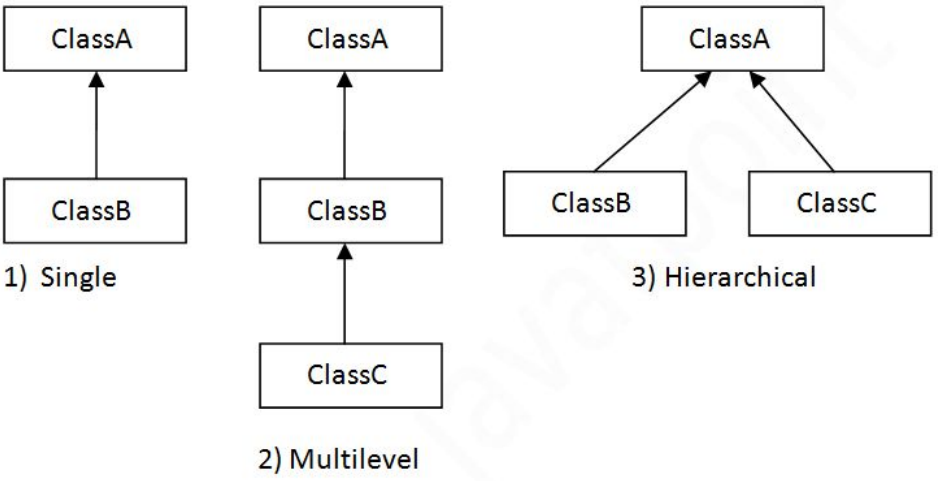
```
$ javac SimpleInheritance.java
$ java SimpleInheritance
show of parent class A
Show of child class B
show of parent class A
```

Types of inheritance in java

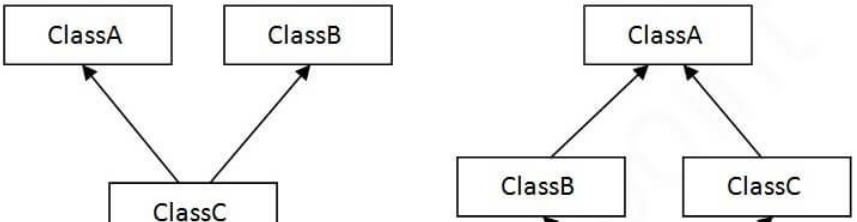
On the basis of class, there can be three types of inheritance in java:

- single,
- multilevel and
- hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only.



When one class inherits multiple classes, it is known as multiple inheritance. Multiple inheritance is not supported in Java through class.



Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

```
class Animal
{
        void eat()
        {
                System.out.println("eating...");
        }
}
class Dog extends Animal
{
        void bark()
        {
                System.out.println("barking...");
        }
}
class TestInheritance
{
        public static void main(String args[])
        {
                Dog d=new Dog();
                d.bark();
                d.eat();
        }
}
```

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

```
class Animal
{
        void eat()
        {
                System.out.println("eating...");
```

```
	}
}

class Dog extends Animal{
	void bark(){
		System.out.println("barking...");
	}
}

class BabyDog extends Dog{
	void weep(){
		System.out.println("weeping...");
	}
}

class TestInheritance2{
	public static void main(String args[]){
		BabyDog d=new BabyDog();
		d.weep();
		d.bark();
		d.eat();
	}
}
```

Hierarchical Inheritance Example

When two or more classes inherit a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherit the Animal class, so there is hierarchical inheritance.

```
class Animal{
	void eat(){
		System.out.println("eating...");
	}
}

class Dog extends Animal{
	void bark(){
		System.out.println("barking...");
	}
}

class Cat extends Animal{
	void meow(){
		System.out.println("meowing...");
	}
}

class TestInheritance3{
	public static void main(String args[]){
		Cat c=new Cat();
		c.meow();
```

```
                c.eat();
                //c.bark();       //C.T.Error
        }
}
```

c. Explain dynamic method dispatch in JAVA. (07 Marks)

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

A superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Here is an example that illustrates dynamic method dispatch:

```
// Dynamic Method Dispatch
class A {
        void callme() {
                System.out.println("Inside A's callme method");
        }
}

class B extends A {
        // override callme()
        void callme() {
                System.out.println("Inside B's callme method");
        }
}

class C extends A {
        // override callme()
        void callme() {
                System.out.println("Inside C's callme method");
        }
}

class Dispatch {
        public static void main(String args[]) {
                A a = new A(); // object of type A
                B b = new B(); // object of type B
                C c = new C(); // object of type C
                A r; // obtain a reference of type A
                r = a; // r refers to an A object
```

```
            r.callme(); // calls A's version of callme
            r = b; // r refers to a B object
            r.callme(); // calls B's version of callme
            r = c; // r refers to a C object
            r.callme(); // calls C's version of callme
        }
}
```

The output from the program is shown here:

Inside A's callme method
Inside B's callme method
Inside C's callme method

This program creates one superclass called A and two subclasses of it, called B and C. Subclasses B and C override callme( ) declared in A. Inside the main( ) method, objects of type A, B, and C are declared. Also, a reference of type A, called r, is declared. The program then in turn assigns a reference to each type of object to r and uses that reference to invoke callme( ). As the output shows, the version of callme( ) executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, r, you would see three calls to A's callme( ) method.

Module-4

7 a. What is an exception? Write the syntax of try and catch block to handle multiple exception. (07 Marks)

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed.

Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed via five keywords

1. try
2. catch
3. throw
4. finally
5. throws

try:

◦ Program statements that you want to monitor for exceptions are placed within the try block.
◦ If an exception occurs within the try block, it is thrown

catch:
The exception thrown in the try block is caught in the catch block. Thus the catch clause catches an exception and handles it in some rational manner. A catch clause must be placed immediately following a try block. We need to specify the exception type that we wish to catch in the catch clause.

This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed after try block ends
}
```
Here, ExceptionType is the type of exception that has occurred.

b. Explain throw and throws keyword. (06 Marks)

throw:
◦ The keyword throw is used to manually throw an exception.
◦ Syntax: throw ThrowableInstance;
◦ where ThrowableInstance is an object of type Throwable or a subclass of Throwable.
◦ There are two ways to obtain a Throwable object
1. using a parameter in a catch clause
2. creating one with the new operator

throws:
◦ If a method is causing an exception that it does not handle then it must specify this behavior so that callers of the method can guard themselves against that exception. We do this by including a throws clause in the method's declaration.
◦ A throws clause lists the type of exceptions that a method might throw. The general form of a method declaration that includes a throws clause

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

```
// Java program to demonstrate working of throw and throws
class ThrowsExecp
{
   static void fun() throws IllegalAccessException
   {
      System.out.println("Inside fun(). ");
      throw new IllegalAccessException("demo");
   }
   // This is a caller function
   public static void main(String args [ ])
   {
      try
      {
         fun();
      }
      catch (IllegalAccessException e)
      {
         System.out.println("caught in main.");
      }
   }
}
```

Output:

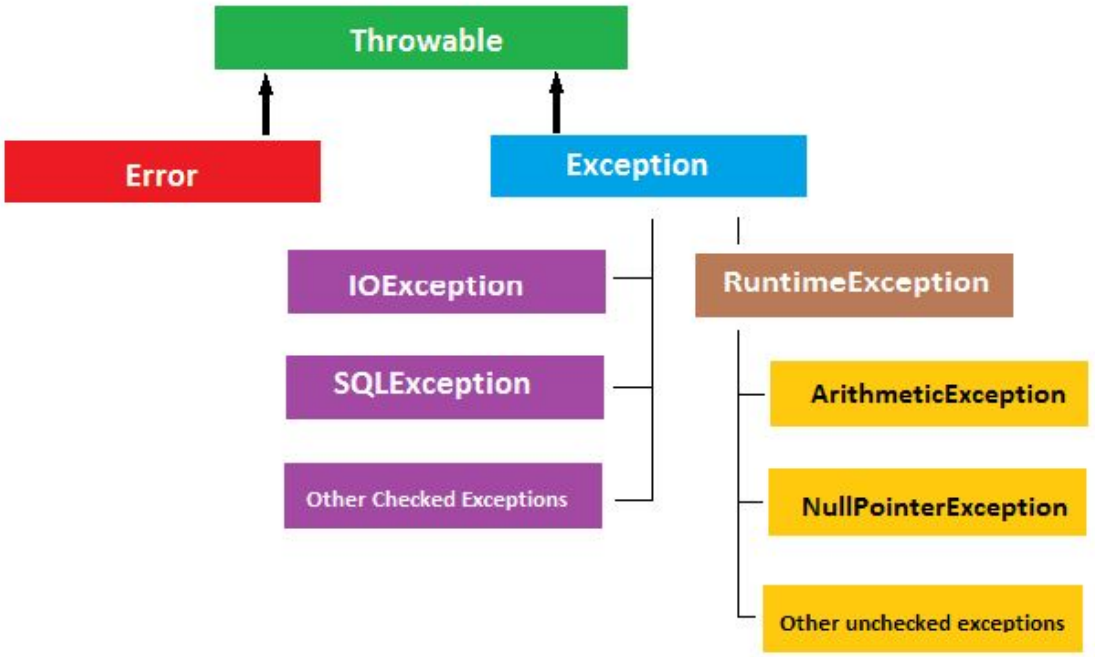
```
Inside fun().
caught in main.
```

c. Explain different types of exception. (07 Marks)

Exception Types:

- All exception types are subclasses of the built-in class **Throwable**.
- **Throwable** is at the top of the exception class hierarchy.
- Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches.
- One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch.

- There is an important subclass of **Exception** called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.
- The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program.
- Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment itself. Stack overflow is an example of such an error.
- The figure below shows the pictorial representation of Exception types.



8 a. Write a note on nested try statements. (07 Marks)

The try statement can be nested. That is, a try statement can be inside the block of another try. Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception.

Here is an example that uses nested try statements:

```
// An example of nested try statements.
class NestTry {
        public static void main(String args[]) {
                try {
                        int a = args.length;
                        /* If no command-line args are present,
                                the following statement will generate
                                a divide-by-zero exception.
                        */
                        int b = 42 / a;
                        System.out.println("a = " + a);
```

```
                try { // nested try block
                        /* If one command-line arg is used,
                                then a divide-by-zero exception
                                will be generated by the following code.
                        */
                        if(a==1) a = a/(a-a); // division by zero
                        /* If two command-line args are used,
                        then generate an out-of-bounds exception. */
                        if(a==2) {
                                int c[] = { 1 };
                        }
                }
                catch(ArrayIndexOutOfBoundsException e) {
                        System.out.println("Array index out-of-bounds: " + e);
                }
        }
        catch(ArithmeticException e) {
                System.out.println("Divide by 0: " + e);
        }
    }
}
```

The program works as follows. When you execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer try block. Execution of the program with one command-line argument generates a divide-by-zero exception from within the nested try block. Since the inner block does not catch this exception, it is passed on to the outer try block, where it is handled. If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner try block.

Here are sample runs that illustrate each case:

```
$java NestTry
Divide by 0: java.lang.ArithmeticException: / by zero

$java NestTry One
a = 1
Divide by 0: java.lang.ArithmeticException: / by zero

$java NestTry One Two
a = 2
Array index out-of-bounds:
java.lang.ArrayIndexOutOfBoundsException:42
```

b. Briefly explain Java's built-in exception. (06 Marks)

Java's built in Exceptions:

- Inside the standard package **java.lang**, Java defines several exception classes.
- Most of these exceptions are subclasses of the standard type **RuntimeException**.
- Exceptions of type RuntimeException need not be included in any method's **throws** list. They are called **unchecked exceptions** because the compiler does not check to see if a method handles or throws these exceptions. The list of unchecked exceptions is shown below.

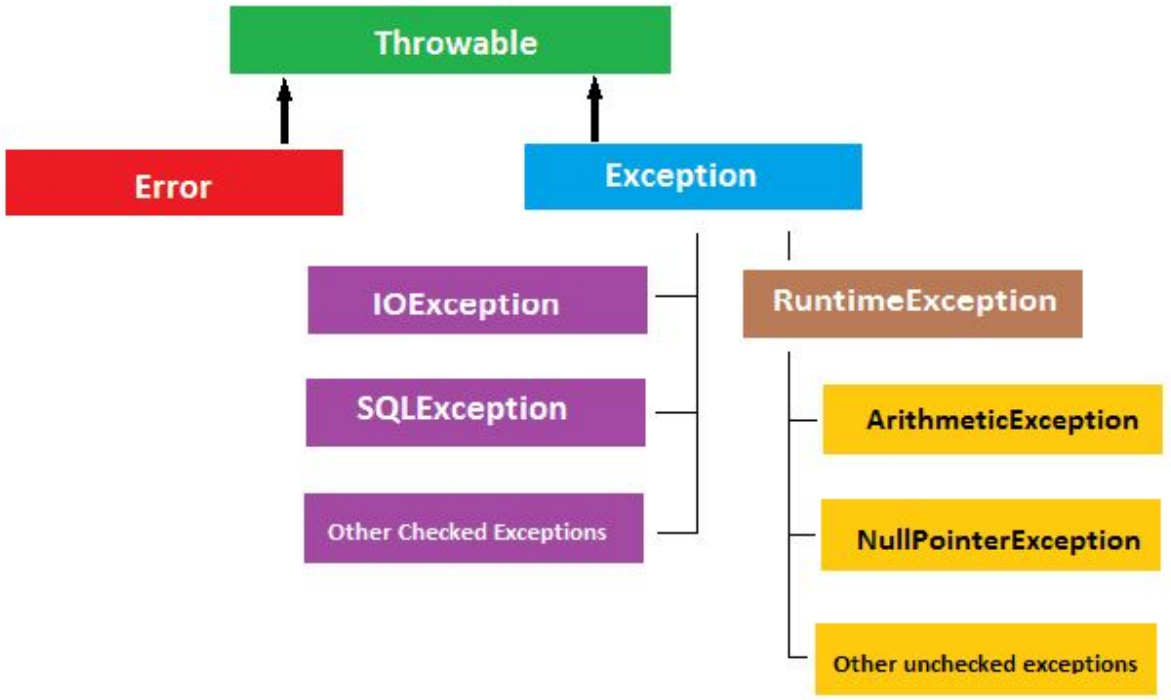
- The list of **checked exceptions** is also shown below. Checked exceptions are exceptions defined by java.lang that must be included in a method's throws list if that method can generate one of those exceptions and does not handle it itself.

The following are the Java's Unchecked RuntimeException Subclasses defined in java.lang-

1. ArithmeticException - Arithmetic error such as divide by zero
2. ArrayIndexOutOfBoundsException – Array index is out of bounds
3 ArrayStoreException – Assignment to an array element of an incompatible type
4. ClassCastException – Invalid Cast
5. EnumConstantNotPresentException – An attempt is made to use an undefined enumeration value.
6. IllegalArgumentException – Illegal argument used to invoke a method
7. IllegalMonitorStateException – Illegal monitor operation, such as waiting on an unlocked thread
8. IllegalStateException – Environment or application is in incorrect state
9. IllegalThreadStateException – Requested operation not compatible with current thread state
10. IndexOutOfBoundsExcepton – Some type of index is out of bounds
11. NegativeArraySizeException – Array created with a negative size
12. NullPointerException – Invalid use of a null reference
13. NumberFormatException – Invalid conversion of a string to a numeric format
14. SecurityException – Attempt to violate security
15. StringIndexOutOfBounds – Attempt to index outside of bounds of a string
16. TypeNotPresentException – Type not found
17. UnsupportedOperationException – An unsupported operation was encountered

Java's Checked Exceptions Defined in java.lang

1. ClassNotFoundException – Class not found
2. CloneNotSupportedExcepton – Attempt to clone an object that does not implement the cloneable interface
3. IllegalAccessException – Access to a class is denied
4. InstantiationException – Attempt to create an object of an abstract class or interface
5. InterruptedException – One thread has been interrupted by another thread.
6. NoSuchFieldException – A requested field does not exist
7. NoSuchMethodException – A requested method does not exist



The figure shows the exception hierarchy in Java

c. What are packages in Java? Explain. (07 Marks)

Packages are containers for classes that are used to keep the class name space compartmentalized. For example, a package allows the user to create a class named **List**, which can be stored in user's own package without concern that it will collide with some other class named **List** stored elsewhere. **Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.** In the absence of packages, you have to use a unique name for each class to avoid name collisions. After a while, you could run out of convenient, descriptive names for individual classes. You also need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers. (Imagine a small group of programmers developing an application)

Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

DEFINING A PACKAGE

T**o create a package simply include a package command as the first statement in a Java source file.** Any classes declared within that file will belong to the specified package.

The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the default package, which has no name.
While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code. The general form of the package statement:

package pkg;

Here, **pkg** is the name of the package.
For example, the following statement creates a package called MyPackage.

package MyPackage;

Java uses file system directories to store packages. For example, the .class files for any classes you declare to be part of MyPackage must be stored in a directory called MyPackage. Remember that case is significant, and the directory name must match the package name exactly.

More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

package pkg1[.pkg2[.pkg3]];

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

package java.awt.image;

needs to be stored in java\awt\image in a Windows environment. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH

Packages are mirrored by directories. This raises an important question:

How does the Java run-time system know where to look for packages that you create? The answer has three parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the CLASSPATH environment variable. Third, you can use the -classpath option with java and javac to specify the path to your classes.

For example, consider the following package specification:

package MyPack

In order for a program to find MyPack, one of three things must be true. Either the program can be executed from a directory immediately above MyPack, or the CLASSPATH must be

set to include the path to MyPack, or the -classpath option must specify the path to MyPack when the program is run via java.

When the second two options are used, the class path must not include MyPack, itself.

It must simply specify the path to MyPack. For example, in a Windows environment, if the path to MyPack is

C:\MyPrograms\Java\MyPack

Then the class path to MyPack is

C:\MyPrograms\Java

The easiest way to try the examples shown in this book is to simply create the package directories below your current development directory, put the .class files into the appropriate directories, and then execute the programs from the development directory. This is the approach used in the following example.

Module-5

9 a. What is an applet? With a skeleton code explain methods that constitute life cycle of applet. (07 Marks)

APPLET:

An applet is a Java code that must be executed within another program. It mostly executes in a Java-enabled web browser.
Applets are dynamic and interactive programs. They are usually small in size and facilitate event-driven applications that can be transported over the web.

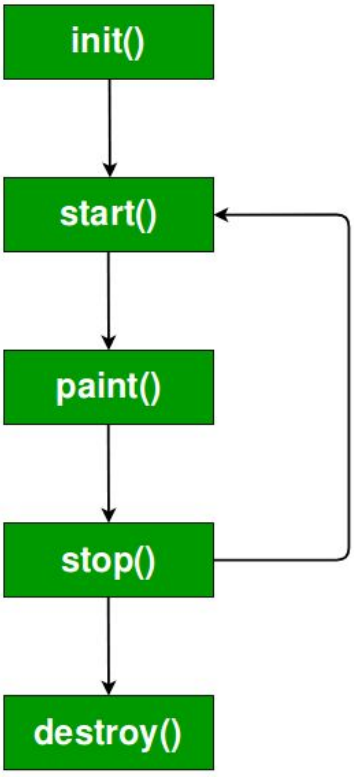
AN APPLET SKELETON:
In general, applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution.
Four of these methods, init(), start(), stop() and destroy() apply to all applets and are defined by Applet.
Default implementations for all of these methods are provided.
Applets do not need to override those methods they do not use.

These five methods can be assembled into the skeleton as shown below -

```
// An Applet skeleton
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkeleton" width=500 height=300>
</applet>
*/

public class AppletSkeleton extends Applet {
// Called first.
public void init() {
// initialization
}

// Called second, after init().
// Also called whenever the applet is restarted
public void start() {
// start or resume execution
}

// Called when the applet is stopped
public void stop() {
// suspends execution
}

// called when applet is terminated.

// This is the last method executed
public void destroy () {
// perform shutdown activities
}

// Called when an applets's window must be restored.
public void paint(Graphics g) {
// redisplay contents of window
}
```

}

The following is the order in which the various methods shown in the skeleton are called
When the applet begins, the following methods are called, in this sequence
1. init()
2. start()
3. paint()
when an applet is terminated, the following sequence of method calls takes place
1. stop()
2. destroy()



init():
- The init() method is the first method to be called.
- We need to initialize variables here.
- This method is called only once during the run time of applet.

start():
- The start method is called after init().
- It is also called to restart an applet after it has been stopped.
- start() is called each time an applet's HTML document is displayed onscreen.
- So, it a user leaves a web page and comes back, the applet resumes execution at start().

paint():
- The paint() method is called each time your applet's output must be redrawn.
- For example, when the applet window may be minimized and then restored.
- Paint() must also be called when the applet begins execution.
- For any reason, whenever the applet must redraw its output, paint() is called.
- The paint() method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running.

stop():

- The stop() method is called when a web browser leaves the HTML document containing the applet. For example, when it goes to another page.
- When stop() is called, the applet is probably running.
- We should use stop() to suspend threads that don't need to run when the applet is not visible.
- We can restart them when start() is called if the user returns to the page.

destroy():

- The destroy() method is called when the environment determines that your applet need to be removed completely from memory.
- At this point, we should free up any resources the applet may be using.
- The stop() method is always called before destroy().

b. List drawbacks of AWT. Explain two swing features. (07 Marks)

Draw backs of AWT are

- AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface.
- Look and feel of the component is defined by the platform, not by Java.
- AWT components use native code resources hence they are referred to as heavy weight
- AWT need HTML code to run the applet
- Doesn't use MVC

Two Key Swing Features

Two key swing features are

- lightweight components and
- a pluggable look and feel.

Swing Components Are Lightweight

With very few exceptions, Swing components are lightweight. This means that they are written entirely in Java and do not map directly to platform-specific peers. Because lightweight components are rendered using graphics primitives, they can be transparent, which enables nonrectangular shapes. Thus, lightweight components are more efficient and more flexible. Furthermore, because lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system. This means that each component will work in a consistent manner across all platforms.

Swing Supports a Pluggable Look and Feel

Swing supports a pluggable look and feel (PLAF). Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing. This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does.

Separating out the look and feel provides a significant advantage: it becomes possible to change the way that a component is rendered without affecting any of its other aspects. In other words, it is possible to "plug in" a new look and feel for any given component without creating any side effects in the code that uses that component. Moreover, it becomes possible to define entire sets of look-and-feels that represent different GUI styles. To use a specific style, its look and feel is simply "plugged in." Once this is done, all components are automatically rendered using that style.

Pluggable look-and-feels offer several important advantages. It is possible to define a look and feel that is consistent across all platforms. Conversely, it is possible to create a look nd feel that acts like

a specific platform. For example, if you know that an application will be running only in a Windows environment, it is possible to specify the Windows look and feel. It is also possible to design a custom look and feel. Finally, the look and feel can be changed dynamically at run time.

c. Explain briefly swing buttons with code snippets. (06 Marks)

The Swing Buttons
Swing defines four types of buttons: JButton, JToggleButton, JCheckBox, and JradioButton. All are subclasses of the AbstractButton class, which extends JComponent. Thus, all buttons share a set of common traits.

AbstractButton contains many methods that allow you to control the behavior of buttons. For example, you can define different icons that are displayed for the button when it is disabled, pressed, or selected. Another icon can be used as a rollover icon, which is displayed when the mouse is positioned over a button. The following methods set these icons:

void setDisabledIcon(Icon di)
void setPressedIcon(Icon pi)
void setSelectedIcon(Icon si)
void setRolloverIcon(Icon ri)

Here, di, pi, si, and ri are the icons to be used for the indicated purpose.
The text associated with a button can be read and written via the following methods:

String getText( )
void setText(String str)

Here, str is the text to be associated with the button.

The model used by all buttons is defined by the ButtonModel interface. A button generates an action event when it is pressed. Other events are possible. Each of the concrete button classes is examined next.

JButton
The JButton class provides the functionality of a push button. You have already seen a simple form of it in the preceding chapter. JButton allows an icon, a string, or both to be associated with the push button. Three of its constructors are shown here:

JButton(Icon icon)
JButton(String str)
JButton(String str, Icon icon)

Here, str and icon are the string and icon used for the button.

When the button is pressed, an ActionEvent is generated. Using the ActionEvent object passed to the actionPerformed( ) method of the registered ActionListener, you can obtain the action command

string associated with the button. By default, this is the string displayed inside the button. However, you can set the action command by calling setActionCommand( ) on the button. You can obtain the action command by calling getActionCommand( ) on the event object. It is declared like this:

String getActionCommand( )

The action command identifies the button. Thus, when using two or more buttons within the same application, the action command gives you an easy way to determine which button was pressed.

In the preceding chapter, you saw an example of a text-based button. The following demonstrates an icon-based button. It displays four push buttons and a label. Each button displays an icon that represents the flag of a country. When a button is pressed, the name of that country is displayed in the label.

```
// Demonstrate an icon-based JButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JButtonDemo" width=250 height=450>
</applet>
*/
public class JButtonDemo extends JApplet
implements ActionListener {
JLabel jlab;
public void init() {
try {
SwingUtilities.invokeAndWait(
new Runnable() {
public void run() {
makeGUI();
}
}
);
} catch (Exception exc) {
System.out.println("Can't create because of " + exc);
}
}

private void makeGUI() {
// Change to flow layout.
setLayout(new FlowLayout());
// Add buttons to content pane.
ImageIcon france = new ImageIcon("france.gif");
JButton jb = new JButton(france);
jb.setActionCommand("France");
jb.addActionListener(this);
add(jb);
ImageIcon germany = new ImageIcon("germany.gif");
jb = new JButton(germany);
jb.setActionCommand("Germany");
```

```
jb.addActionListener(this);
add(jb);
ImageIcon italy = new ImageIcon("italy.gif");
jb = new JButton(italy);
jb.setActionCommand("Italy");
jb.addActionListener(this);
add(jb);
ImageIcon japan = new ImageIcon("japan.gif");
jb = new JButton(japan);
jb.setActionCommand("Japan");
jb.addActionListener(this);
add(jb);
// Create and add the label to content pane.
jlab = new JLabel("Choose a Flag");
add(jlab);
}
// Handle button events.
public void actionPerformed(ActionEvent ae) {
jlab.setText("You selected " + ae.getActionCommand());
}
}
```

JToggleButton

A useful variation on the push button is called a toggle button. A toggle button looks just like a push button, but it acts differently because it has two states: pushed and released. That is, when you press a toggle button, it stays pressed rather than popping back up as a regular push button does. When you press the toggle button a second time, it releases (pops up). Therefore, each time a toggle button is pushed, it toggles between its two states.

Toggle buttons are objects of the JtoggleButton class. JToggleButton implements AbstractButton. In addition to creating standard toggle buttons, JToggleButton is a superclass for two other Swing components that also represent two-state controls. These are JCheckBox and JRadioButton, which are described later in this chapter. Thus, JtoggleButton defines the basic functionality of all two-state components.

JToggleButton defines several constructors. The one used by the example in this section is shown here:

JToggleButton(String str)

This creates a toggle button that contains the text passed in str. By default, the button is in the off position. Other constructors enable you to create toggle buttons that contain images, or images and text.

JToggleButton uses a model defined by a nested class called JToggleButton.ToggleButtonModel. Normally, you won't need to interact directly with the model to use a standard toggle button.

Like JButton, JToggleButton generates an action event each time it is pressed. Unlike JButton, however, JToggleButton also generates an item event. This event is used by those components that support the concept of selection. When a JToggleButton is pressed in, it is selected. When it is popped out, it is deselected.

To handle item events, you must implement the ItemListener interface. Each time an item event is generated, it is passed to the itemStateChanged( ) method defined by ItemListener. Inside itemStateChanged( ), the getItem( ) method can be called on the ItemEvent object to obtain a reference to the JToggleButton instance that generated the event. It is shown here:

Object getItem( )

A reference to the button is returned. You will need to cast this reference to JtoggleButton. The easiest way to determine a toggle button's state is by calling the isSelected( ) method (inherited from AbstractButton) on the button that generated the event. It is shown here:

boolean isSelected( )

It returns true if the button is selected and false otherwise.

Here is an example that uses a toggle button. Notice how the item listener works. It simply calls isSelected( ) to determine the button's state.

```
// Demonstrate JToggleButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JToggleButtonDemo" width=200 height=80>
</applet>
*/
public class JToggleButtonDemo extends JApplet {
JLabel jlab;
JToggleButton jtbn;
public void init() {
try {
SwingUtilities.invokeAndWait(
new Runnable() {
public void run() {
makeGUI();
}
}
);
} catch (Exception exc) {
System.out.println("Can't create because of " + exc);
}
}
private void makeGUI() {
// Change to flow layout.
setLayout(new FlowLayout());
// Create a label.
jlab = new JLabel("Button is off.");
// Make a toggle button.
jtbn = new JToggleButton("On/Off");
// Add an item listener for the toggle button.
```

```
jtbn.addItemListener(new ItemListener() {
public void itemStateChanged(ItemEvent ie) {
if(jtbn.isSelected())
jlab.setText("Button is on.");
else
jlab.setText("Button is off.");
}
});
// Add the toggle button and label to the content pane.
add(jtbn);
add(jlab);
}
}
```

10 a. Explain components and containers in swings. (07 Marks)

Components

In general, Swing components are derived from the JComponent class. (The only exceptions to this are the four top-level containers, described in the next section.) JComponent provides the functionality that is common to all components. For example, JComponent supports the pluggable look and feel. JComponent inherits the AWT classes Container and Component. Thus, a Swing component is built on and compatible with an AWT component.

All of Swing's components are represented by classes defined within the package javax.swing. The following table shows the class names for Swing components (including those used as containers).

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayeredPane
JList	JMenu	JMenuBar	JMenuItem
JOptionPane	JPanel	JPasswordField	JPopupMenu
JProgressBar	JRadioButton	JRadioButtonMenuItem	JRootPane
JScrollBar	JScrollPane	JSeparator	JSlider
JSpinner	JSplitPane	JTabbedPane	JTable
JTextArea	JTextField	JTextPane	JTogglebutton
JToolBar	JToolTip	JTree	JViewport
JWindow			

Containers

Swing defines two types of containers.

The first are top-level containers: JFrame, Japplet, JWindow, and JDialog. These containers do not inherit JComponent. They do, however, inherit the AWT classes Component and Container. Unlike Swing's other components, which are lightweight, the top-level containers are heavyweight. This makes the top-level containers a special case in the Swing component library.

As the name implies, a top-level container must be at the top of a containment hierarchy. A top-level container is not contained within any other container. Furthermore, every containment hierarchy must begin with a top-level container. The one most commonly used for applications is JFrame. The one used for applets is JApplet.

The second type of containers supported by Swing are lightweight containers. Lightweight containers do inherit JComponent. An example of a lightweight container is JPanel, which is a general-purpose container. Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container. Thus, you can use lightweight containers such as JPanel to create subgroups of related controls that are contained within an outer container.

b. Explain different string operations in JAVA. (07 Marks)

Different string operatons in JAVA are
1. String concatenation
2. Character Extraction
3. String comparision
4. Searching strings
5. Modifying string
6. Changing the case of characters within string

1. 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. This allows you to chain together a series of + operations.

For example, the following fragment concatenates three strings:

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

This displays the string “He is 9 years old.”

String Conversion and toString( )

The toString( ) method returns is the string representation for objects of classes.

The toString( ) method has this general form:

```
String toString( )
```

To implement toString( ), simply return a String object that contains the human-readable string that appropriately describes an object of your class.

2. Character Extraction

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 where)

Here, where is the index of the character that you want to obtain. The value of where must be nonnegative and specify a location within the string. charAt( ) returns the character at the specified location.

For example,
char ch;
ch = "abc".charAt(1);
assigns the value “b” to ch.

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)

Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from sourceStart through sourceEnd–1. The array that will receive the characters is specified by target. The index within target at which the substring will be copied is passed in targetStart. Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring.

getBytes( )
There is an alternative to getChars( ) that stores the characters in an array of bytes. This method is called getBytes( ), and it uses the default character-to-byte conversions provided by the platform.

Here is its simplest form:
byte[ ] getBytes( )

Other forms of getBytes( ) are also available. getBytes( ) is most useful when you are exporting a String value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

toCharArray( )
If you want to convert all the characters in a String object into a character array, the easiest way is to call toCharArray( ). It returns an array of characters for the entire string. It has this

general form:
char[ ] toCharArray( )

This function is provided as a convenience, since it is possible to use getChars( ) to achieve the same result.

3. String Comparison
The String class includes several 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.

regionMatches( )
The regionMatches( ) method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows you to ignore case in such comparisons.

Here are the general forms for these two methods:
boolean regionMatches(int startIndex, String str2,
int str2StartIndex, int numChar

boolean regionMatches(boolean ignoreCase,
int startIndex, String str2,
int str2StartIndex, int numChars)

For both versions, startIndex specifies the index at which the region begins within the invoking String object. The String being compared is specified by str2. The index at which the comparison will start within str2 is specified by str2StartIndex. The length of the substring being compared is passed in numChars. In the second version, if ignoreCase is true, the case of the characters is ignored. Otherwise, case is significant.

startsWith( ) and endsWith( )
String defines two routines that are, more or less, specialized forms of regionMatches( ). 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)

Here, str is the String being tested. If the string matches, true is returned. Otherwise, false
is returned. For example,
"Foobar".endsWith("bar")
and
"Foobar".startsWith("Foo")
are both true.

equals( ) Versus ==
It is important to understand that the equals( ) method and the == operator perform two different operations. As just explained, 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. The

following program shows how two different String objects can contain the same characters, but references to these objects will not compare as equal:

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

compareTo( )

Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is less than, equal to, or greater than the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The String method compareTo( ) serves this purpose.

It has this general form:

```
int compareTo(String str)
```

4. Searching Strings

The String class provides two methods that allow you to search a string for a specified character or substring:

- indexOf( ) Searches for the first occurrence of a character or substring.
- lastIndexOf( ) Searches for the last occurrence of a character or substring.

These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or 1 – on failure.

To search for the first occurrence of a character, use

```
int indexOf(int ch)
```

To search for the last occurrence of a character, use

```
int lastIndexOf(int ch)
```

Here, ch is the character being sought.

To search for the first or last occurrence of a substring, use

```
int indexOf(String str)
int lastIndexOf(String str)
```

Here, str specifies the substring.

You can specify a starting point for the search using these forms:

```
int indexOf(int ch, int startIndex)
int lastIndexOf(int ch, int startIndex)
int indexOf(String str, int startIndex)
int lastIndexOf(String str, int startIndex)
```

Here, startIndex specifies the index at which point the search begins. For indexOf( ), the search runs from startIndex to the end of the string. For lastIndexOf( ), the search runs from startIndex to zero.

5. Modifying a String

Because String objects are immutable, whenever you want to modify a String, you must either copy it into a StringBuffer or StringBuilder, or use one of the following String methods, which will construct a new copy of the string with your modifications complete.

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.
The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

concat( )

You can concatenate two strings using concat( ), shown here:

String concat(String str)

This method creates a new object that contains the invoking string with the contents of str appended to the end. concat( ) performs the same function as +. For example,

```
String s1 = "one";
String s2 = s1.concat("two");
```

puts the string “onetwo” into s2. It generates the same result as the following sequence:

```
String s1 = "one";
String s2 = s1 + "two";
```

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( )

Here is an example:
String s = " Hello World ".trim();

This puts the string “Hello World” into s.

6. 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 general forms of these methods:

String toLowerCase( )
String toUpperCase( )
Both methods return a String object that contains the uppercase or lowercase equivalent of the invoking String.

c. Wtite a program in Java to search a key string (06 Marks)

```
class SearchKey
{
	public static void main(String args[])
	{
		String
s[]={"Now","is","the","time","for","all","good","men","to","come","to","the","aid","of","their","co
untry"};
			String key="Country";
			boolean flag=false;
		for(int i=0;i<s.length-1;i++)
		{
			for(int j=0;j<s.length-1-i;j++)
			{
				if(s[j].compareTo(s[j+1])>0) // compare s[j] with s[j+1]
				{
					String temp=s[j];
					s[j]=s[j+1];
					s[j+1]=temp;
```

```
                    }
                }
            }
            for(int i=0;i<s.length-1;i++)
            {
                if (s[i].compareTo(key) == 0)
                {
                    flag=true;
                    break;
                }
            }
            if(flag == true)
            {
                System.out.println("Key found ");
            }
            else
            {
                System.out.println("Key not found");
            }
        }
}
```