1 (a). List and explain the java buzzwords. [CO1, L1, L4] 10 Marks

The following is the list of Java buzzwords
1. Simple
2. Secure
3. Portable
4. Object-Oriented
5. Robust
6. Multithreaded
7. Architecture-neutral
8. Interpreted
9. High performance
10. Distributed
11. Dynamic

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 multiprocess 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.

7. Architecture-neutral
The main issue for the Java designers was that of **code longevity and portability**. One of the main concerns of programmers is that there is no guarantee that their program will run tomorrow even on the same system. Operating system upgrades, processor upgrades and changes in core system resources together make a program malfunction. Java is been designed with the goal **"write once and run anywhere, anytime, forever",** and to a great extent this goal is accomplished.

8. Interpreted and High Performance:
Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using **just-in-time compiler.**

9. Distributed:
As C is to system programming, Java is to Internet programming. **Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols.** Accessing a resource using a URL is not much different from accessing a file. Java also supports **Remote Method Invocation (RMI).** This feature enables a program to invoke methods across a network.

10. Dynamic:
Java programs carry with them substantial amount of run-time type information that is used to verify and resolve accesses to objects at run-time. This makes it possible to dynamically link code in a safe manner. Small fragments of bytecode may be dynamically updated on a running system.

2(a). Discuss three OOP principles with example segment of code [CO1, L2]
10 Marks

The three OOP principles are –

1. Encapsulation
2. Inheritance
3. Polymorphism

1. Encapsulation:
Encapsulation is the mechanism that binds together code and data it manipulates, and keeps both safe from outside interference and misuse.
Encapsulation is wrapping of data and function or method into a single unit.
Encapsulation is a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well defined interface.
The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details and without fear of unexpected side effects.
In Java the basis of encapsulation is the class. A class defines the structure and behaviour (data and code) that will be shared by a set of objects. **Objects are referred to as instances of a class.** Thus, a class is a logical construct; an object has physical reality. (Class is like a blue print of a building and object is the real building).
The code and data that constitute a class is collectively called **members of the class**. **The data are referred to as member variables or instance variables. The code that operates on the data is referred to as member methods or just methods.** Methods define how the member variables can be used. That is, the behaviour and interface of a class are defined by the methods that operate on its instance data.
There are mechanisms for hiding the complexity of the implementation inside the class because the purpose of the class is to encapsulate complexity. Each member or variable in a class can be marked **private or public**. The **public interface** of a class represents everything that external users of the class need to know. The **private methods and data** can only be accessed by code that is a member of the class. Any other code that is not a member of the class cannot access a private method or variable.

2. Inheritance:
Inheritance is the process by which one object acquires the properties of another object.
Inheritance supports the concept of hierarchical classification. For example, a Golden Retriever belongs to the class - **dog**, dog inturn is part of the class **mammal**, and mammal is under the larger class **animal**. Mammal is called the subclass of animals and animals is called the mammal's superclass.

Without inheritance, each object has to define all of its characteristics explicitly. But, by use of inheritance, an object needs to define only those qualities that make it unique within its class. It inherits its general attributes from its parent. Therefore, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

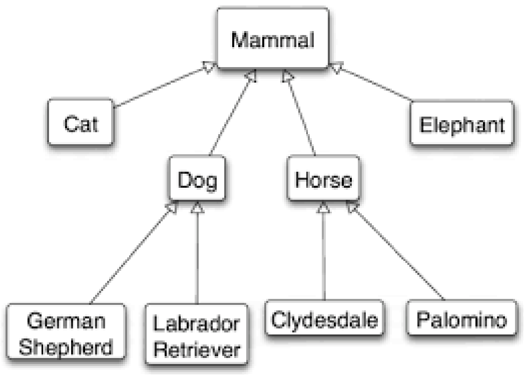

Fig: Animal Kingdom – Example for Inheritance

Inheritance interacts with encapsulation. If a given class encapsulates some attributes, then any subclass will have the same attributes plus any that it adds as part of its specialization. It is this key concept that lets object-oriented programs grow in complexity linearly rather than geometrically. A new sub-class inherits all of the attributes of all of its ancestors. It does not have unpredictable interactions with the majority of the rest of the code in the system.

3. Polymorphism:

Polymorphism in Greek means "**many forms**".

It is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.

For example, consider a program to implement three types of stacks, say, one for integer values, one for floating-point values and one for characters. The algorithm that implements each stack is the same, but the data stored is different. In a process oriented model we have to create three different stack routines each with different names. However, because of polymorphism, in Java we can create a general set of stack routines, all having the same name.

The concept of polymorphism is expressed by the phrase **"one interface, multiple methods."** It means that it is possible to design a generic interface to a group of related activities.

Polymorphism helps reduce complexity by allowing the same interface to be used to specify a general class of action.

It is the compiler's job to select the specific action (or interface or method) as it applies to each situation. The programmer need not select the method manually.

3 (a). Briefly explain primitive types in Java. Demonstrate type casting with the help of a program [CO2, L5] 10 Marks

1. The Primtive Data Types:

Primitive data types are typically types that are **built-in or basic** to a language implementation.

Primitive types are also commonly referred to as simple types or basic types.

Primitive data types are used to construct **arrays and class** types.

They form the basis for all other types of data that a programmer can create.

The primitive data types are defined to have explicit range and mathematical behavior.

Languages such as C and C++ allow the size of an integer to vary based upon the particular platform. However, in Java it is different. **Because of Java's portability requirement, all data types have a strictly defined range.** For example, an **int** is always 32 bits, regardless of the particular platform. This allows programs to be written that are guaranteed to run without porting on any machine architecture.

Java defines **eight** primitive types of data. They are -

1. byte
2. short
3. int
4. long
5. char
6. float
7. double
8. boolean

The Primitive data types can be put into four groups -

a) Integers
b) Floating-point numbers
c) Characters
d) Boolean

a) Integers:

Java defines four integer types: **byte, short, int and long.**

All are signed, poisitve and negative values.

Java does not support unsigned, positive-only integers.

The following table shows the width and ranges of the integer types:

#	Data type	Width	Range
1	long	64	-9,223,370,036,854,775,808 to 9,223,372,036,854,775,807
2	int	32	-2,147,483,648 to 2,147,483,647

3	short	16	-32,768 to 32,767
4	byte	8	-128 to 127

byte:

The smallest integer type is **byte**.

This is a signed 8-bit type that has a range from -128 to 127.

Variables of type byte are useful **when you are working with a stream of data from a network or file and when you are working with raw binary data that may not be directly compatible with Java's other built-in types.**

Eg: The following declares two byte variables called b and c.

byte b,c;

short:

short is a signed 16-bit type. It has a range from -32768 to 32,767.

Example - **short s;**

int:

The most commonly used integer type is **int**.

It is signed 32-bit type that has a range from -2,147,483,648 to 2,147,483,647.

Variables of type int are used to control loops and to index arrays. Although we can use byte and short for indexing we use int, because, **when byte and short values are used in an expression they are promoted to int when the expression is evaluated.**

long:

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

Example: Factorial of a number

Floating-point types:

Floating point numbers are **real numbers** and are used when evaluating expressions that require **fractional percision.**

For example, calculations such as square root, sine and cosine result in a value whose precision requires a floating-point type.

Thrre are two kinds of floating point types – **float and double** which represent single and double precision numbers respecitvely.

Their width and ranges are shown in the table below -

#	Data type	Width	Range
1	double	64	$4.9e^{-324}$ to $1.8e^{+308}$
2	float	32	$1.4e^{-045}$ to $3.4e^{+038}$

float:

The type float specifies a single-precision value that uses **32 bits** of storage.

Single precision is faster on some processors and takes half as much as double precision, but will become imprecise when the values are either very large or very small.

Variables of type float are useful when you need a fractional component, but don't require a large degree of precision.

Example:

float hightemp, lowtemp;

double:

double data type uses **64 bits** to store a value.

Double precision is actually faster than sigle precision on some modern processors that have been optimized for high-speed mathematical calculations.

All transcendental math functions, such as sin(), cos() and sqrt() return double values.

Datatype **double** is used when you need to maintain accuracy over many iterative calculations and when you are manipulating large-valued numbers.

Example: To compute the area of the circle

Characters:

The data type used to store characters is **char**.

char in Java is not the same as char in C or C++. In C/C++, char is 8 bits wide but in Java char is 16 bits.

The range of char is 0 to 65536.

There are no negative characters.

The ASCII characters ranges from 0 to 127 and the extended 8-bit character set, ISO-Latin ranges from 0 to 255.

Since Java is designed to allow programs to be written for worldwide use, it makes sense that it would use Unicode to represent characters.

In Java, unlike C/C++, char is 16 bits, because, **Java uses Unicode to represent characters.** Unicode defines a fully international character set that can represent all of the characters found in human languages it is a unification of dozens of characters sets such as Latin, Greek, Arabic, Cycrillic, Hebrew, Katakana, Hangul, and many more. For this purpose it requires 16 bits.

Booleans:

The datatype boolean is used in Java for logical values.

It can have only one of two possible values, **true or false**.

This is the type returned by all relational operators, as in the case of a < b. The conditional expression that govern the control statements such as **if** and **for** also require boolean data type.

Program to demonstrate boolean type

```
// Demonstrate boolean values
class BoolTest
{
        public static void main(String args[])
        {
                boolean b;

                b=false;
                System.out.println("b is " + b);
                b = true;
                System.out.println("b is " + b);

                // a boolean value can control the if statement
                if (b) // if b is true
                System.out.println("This is executed");

                b=false;
                if (b)  // if b is true
                System.out.println("This is not executed");

                // outcome of relational operator is a boolean value
                System.out.println("10 > 9 is " + (10 > 9));
        }
}
```

Output:

```
$ javac BoolTest.java
$ java BoolTest
b is false
b is true
This is executed
10 > 9 is true
$
```

Type Conversion and Casting:

Java's Automatic Conversions:

When one type of data is assigned to another type of variable, an automatic type conversion will take palce 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.

Casting Incompatible Types:

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;
```

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.

4 (a). What is an array in Java, give syntax. [CO2, L1, L3]
i. Write a program to demonstrate one-dimensional array.
ii. Write a program to demonstrate multidimensional array. (2+4+4) Marks

An array is a group of similar data type variables that are referred to by a common name.

Arrays of any type can be created.

Arrays have one or more dimensions.

A specific element in an array is accessed by its index.

Array index starts from zero.

One-Dimensional Arrays:

A one-dimensional array is a list of similar data type variables.

The general form of one-dimensional array is

type var-name[];

The type determines the data type of each element of the array. That is, it determines what type of data the array will hold.

Example:

int month_days[ ];

Even though, the above declaration tells that **month_days** is an array variable, no array exists. The value of **month_days** is set to null, that is, it represents an array with no value.

To link **month_days** with an actual, physical array of integers, we must allocate memory using new and assign it to **month_days.**

new is a special operator that allocates memory.

The syntax to allocate memory using **new** operator is

array-var = new type [size];

Example:

month_days = new int [12];

Now, the **month_days** will refer to an array of 12 integers. **At the same time all the elements in the array will be initialized to zero.**

As we have seen above, obtaining an array is a two step process, first we must declare a variable of the desired array type and secondly we must allocate the memory that will hold the array, using **new operator** and assign it to the array variables. Thus, in Java all arrays are dynamically allocated.

Once the memory allocation is done for the array, we can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero.

The following program demonstrates one dimensional array

```
// Program to demonstrate one-dimensional array
class Array
{
    public static void main(String args[])
    {
          int month_days[];
          month_days = new int[12];
          month_days[0] = 31;
          month_days[1] = 28;
          month_days[2] = 31;
          month_days[3] = 30;
          month_days[4] = 31;
          month_days[5] = 30;
          month_days[6] = 31;
          month_days[7] = 31;
          month_days[8] = 30;
          month_days[9] = 31;
          month_days[10] = 30;
          month_days[11] = 31;
          System.out.println("April has " + month_days[3] + " days. ");
    }
}
```

One step process to define a array:

It is possible to combine the declaration of the array variable with the allocation of the array as shown below :
Syntax:
type array-var [ ] = new type [size];

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

Initialization of one dimensional array:
 Arrays can be initialized when they are declared.
 An array initializer is a list of comma-separated expressions surrounded by curly braces.
 The commas separate the values of the array elements.
 The array will be automatically created large enough to hold the number of elements you specify in the array initializer.
 There is no need to use new.
 Example:

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

TWO 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.
 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 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();
        }
   }
}
```

The above program numbers each element in the array from the left to right, top to bottom, and then displays those values. The output of the program is -

```
0  1  2  3  4
5  6  7  8  9
10  11  12  13  14
15  16  17  18  19
```

When you allocate memory for a multidimensional array, 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.

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
```

INITIALIZATION OF TWO-DIMESIONAL ARRAYS:

The following example demonstrates how to initialize a multi-dimensional array

```
// Program to initialize two dimensionl array
class Matrix
{
   public static void main(String args[])
   {
      int array [ ] [ ] = {
                           { 1, 2, 3, 4 },
                           { 2, 3, 4, 5 },
                           { 3, 4, 5, 6 },
                           { 4, 5, 6, 7 }
                         };
      int i, j;
   for(i=0;i<4;i++)
   {
        for(j=0;j<i+4;j++)
           System.out.print(twoD[i][j] + " ");
        System.out.println();
   }
```

The output of this program is

```
1   2  3   4
2   3  4   5
3   4  5   6
4   5  6   7
```

THREE DIMENSIONAL ARRAYS:

A three dimensional (3D) array can be thought of as an arrray of arrays of arrays.
The first dimentsion represents number of planes or pages.
The second dimension represents the number of rows in each page
The third dimension represents the number of columns in each page
The following program demonstrates a three dimensional array

```
// Demonstrate a three-dimensional array
class ThreeDMatrix
{
    public static void main(String args[])
    {
          int threeD[ ][ ][ ] = new int [3][4][5];
          int i,j,k;

          for (i=0;i<3;i++)
                    for(j=0;j<4;j++)
                              for(k=0;k<5;k++)
                                        threeD[i][j][k] = i * j * k;
          for (i=0;i<3;i++)
          {
                    for(j=0;j<4;j++)
                    {
                              for(k=0;k<5;k++)
                                        System.out.print(threeD[i][j][k] + " " );
                              System.out.println();
                    }
                    System.out.println();
          }
    }
}
```

The output of the program is

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

0 0 0 0 0
```

```
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12

0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24
```

ALTERNATE ARRAY DECLARATION SYNTAX:

There is a second way to declare an array

type [ ] var-name;

Here the square brackets follow the type specifier and not the name of the array variable.

Example:

```
int a1 [ ] = new int [3];
int [ ] a2 = new int [3];
```

Both the declarations are equivalent

The second method offers convenience when declaring several arrays at the same time.

For example:

```
int [ ] num1, num2, num3;   // create three arrays
```

The above declaration is same as

```
int num1[], num2[], num3[];     // create three arrays
```

5 (a). Explain Selection statements and iteration statements with syntax [CO2, L4] 10 Marks

JAVA'S SELECTION STATEMENTS:

Java supports two selection statements: if and switch

Selection statements allow you to control the flow of your program's execution base upon conditins known only during run time.

The if Statement:

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths.

General form

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

Each statement may be a single statement or a comound statement enclosed in curly braces. The condition is any expression that retruns a boolean value. The else clause is optional.

The if works like this: If the condition is true, then statemen1 is executed. Otherwise statement2 (if it exists) is executed. In no case will both statements be executed.

Example:

Nested ifs:

A nested if is an if statement that is the target of another if or else. When we nest ifs, remember that an else statement always refers to the nearest if satement that is within the same block as the else and that is not already associated with an else.

Example:

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

The if-else-if ladder:

General form:

```
if (condition)
     statement;
else if (condition)
     statement;
else if (condition)
     statement;
.
.
.
else
 statement;
```

The **if** statements are executed from top down.

As soon as one of the condtions controlling the **if** is true, the statement associated with that **if** is executed, and the rest of the ladder is bypassed.

If none of the conditions are true, then the final else statement will be executed.

The final else acts as the default condition.

SWITCH:

The **switch** statement is Java’s multiway branch statement.

The general form

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

The expression must be of type byte, short, int or char.

Each of the values specified in the case statements must be of a type compatible with the expression.

An enumeration value can also be used to control a switch statement.

Each case value must be unique literal. Duplicate case vaues are not allowed.

Working: The value of the expression is compared with each of the literal values in the case statements. If a match is found, the code sequence following that case statement is executed. If none of the constants match the value of the expression, then the default statement is executed. The **default** statement is optional. If no case matches and no default is present, then no further action is taken.

The break statement is used inside the switch to terminate a statement sequence. When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement.

If break statement is omitted, execution will continue on into the next **case**.

Nested switch statements:

we can use switch as part of the statement sequence of an outer switch. This is called nested switch.

Important features of the switch statement:

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

No two case constants in the same switch can have identical values. A switch statement and an enclosing outer switch can have case constants in common.

A switch statement is usually more efficient than a set of nested ifs.

When a switch statement is compiled, Java compiler will inspect each of the case constants and create a "jump table" that it will use for selecting the path of execution depending on the value of expression. Therefore, If we want to select among a large group of values, a switch statement will run much faster than the equivalent logic coded using a sequence of if-elses.

Iteration Statements:

Java's iteration statements 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.

Genereal 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 (condtion);
```

Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditinal 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 i sset 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 loopp, 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
```

Nested Loops:
Java allows loops to be nested. That is, one loop may be inside another.
Example

```
class Nested {
     public static void main(String args[ ] )  {
       int i, j;
       for (i = 0; i <10; i ++)  {
                 for (j=i;j<10;j++)
                           System.out.print(“.”);
                 System.out.println();
       }
   }
}
```

Output:

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

6 (a) Create a Java class called Student with USN, Name, Branch, Phone as vairables within it. Write a Java program to create nStudent objects and print the USN, Name, Branch and Phone of these objects with suitable heading. 10 Marks

```
import java.util.Scanner;
public class Student
{
     String USN, Name, Branch;
     long Phoneno;
     public void getinfo()
     {
        Scanner s =new Scanner(System.in);
        System.out.println("Enter USN");
        USN=s.nextLine();
        System.out.println("Enter Name");
        Name=s.nextLine();
        System.out.println("Enter Branch");
        Branch=s.nextLine();
        System.out.println("Enter phone number");
```

```
            Phoneno=s.nextLong();
        }

        public static void main(String[] args)
        {
        int n,i;
        Scanner s=new Scanner(System.in);
        System.out.println("Enter the number of student");
        n=s.nextInt();
        Student[] obj=new Student[n];
        for(i=0;i<n;i++)
        {
                obj[i]=new Student();
                System.out.printf("Student : %d\n",i+1);
                obj[i].getinfo();
        }
        System.out.println("Displaying the details of all the Students");
        for(i=0;i<n;i++)
        {
            System.out.println("Student" + (i+1));
            System.out.println("USN : "+ obj[i].USN);
            System.out.println("Name : "+ obj[i].Name );
            System.out.println("Branch : "+ obj[i].Branch);
            System.out.println("Phone No : "+ obj[i].Phoneno);
        }
 }
}
```

Output:

```
Enter the number of student
2
Student : 1
Enter USN
1cr13cs001
Enter Name
reshma
Enter Branch
cse
Enter phone number
98877676786
Student : 2
Enter USN
1cr13cs002
Enter Name
sherly
Enter Branch
cse
Enter phone number
9876746463
Displaying the details of all the Students
Student1
USN : 1cr13cs001
Name : reshma
Branch : cse
Phone No : 98877676786
Student2
USN : 1cr13cs002
```

Name : sherly
Branch : cse
Phone No : 9876746463

7 (a) Explain Bitwise Operators and briefly explain the ?: operator [CO2, L5] (8+2) marks

THE BITWISE OPERATORS:

Bitwise operators act upon the individual bits of their operands.

Java defines several bitwise operators that can be applied to integer types – long, int, short, char, byte.

The following table shows the list of bitwise operators -

Sl. No.	Operator	Result
1	~	Bitwise unary NOT
2	&	Bitwise AND
3	|	Bitwise OR
4	^	Bitwise exclusive OR
5	>>	Shift Right
6	>>>	Shift Right Zero Fill
7	<<	Shift Left
8	&=	Bitwise AND Assignment
9	|=	Bitwise OR Assignemnt
10	^=	Bitwise exclusive OR Assignment
11	>>=	Shift Right Assignment
12	>>>=	Shift Right Zero Fill Assignment
13	<<=	Shift Left Assignment

Bitwise operators manipulate the bits within an integer. So we need to know how Java stores integers values and how it represents negative numbers.

How poisitive integer values are stored:

All the integer types are represented by binary numbers of varying bits widths.

For example, the byte value of 42 in binary is 00101010– byte size is 8 bits,

MSB LSB

0	0	1	0	1	0	1	0

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

32 + 8 + 2 = 42

How negative integer values are stored:

Java uses an encoding known as two's complement to represent negative numbers in binary format.

To find the binaray equivalent of -42, invert (change 1's to 0's and vice versa) all the bits of +42.

Then add 1 to the result. We get the binary equivalent of -42. The conversion is illustrated below. The binaray equivalent of -42 is 11010110.

The high-order bit determines the sign of an integer. Positive numbers have high order bit 0 and negative numbers have high-order bit 1.

+42

0	0	1	0	1	0	1	0

Invert (change 1's t 0's and vice versa)

1	1	0	1	0	1	0	1

Add 1 +1

we get -42

1	1	0	1	0	1	1	0

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

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.prinln(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
```

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 postions 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.

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.

Bitwise Opearator Compound Assignments:

All of the binary bitwise operators have a compund form similar to that of the algebraic operators, which combines the assignment with the bitwise operation.

For example, the following two statements which shift the value in a right by four bits, are equivalent

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

Similarly,

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

are the same.

The following program demonstrates bitwise operator assignments

```
class OpBitEquals {
    public static void main(String args [ ]) {
        int a =1;
        int b = 2;
        int c = 3;

        a |= 4;
        b  >>= 1;
        c <<= 1;
        a ^= c;
        System.out.println("a = "  a);
        System.out.println("b = "  b);
        System.out.println("c = "  c);
    }
}
```

Output:

```
a = 3
b = 1
c = 6
```

The ? Operator:

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

The operator is ?.

General form:

expression1 ? Expression2 : expression3

expression1 can be any expression that evaluates to a boolean value.
If expression1 is true then expression2 is evaluated, else, expression3 is evaluated.
Example:
ratio = denom == 0 ? 0 : num /denom;
if denom equals zero, then the expression between the question mark and the colon is evaluated and used as the value of the entire ? Expression.
If denom is not equal to zero, then the expression after the colon is evaluated and used for the value of the entire ? Expression.
The result is then assigned to ratio.
Example program

```
class Ternary {
 public static void main(String args[ ]) {
     int i, k;
     i = 10;
     k = i < 0 ? -i : i;
     System.out.println("Absolute value of " + i  + " is " + k);

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

Output:

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