OOC VTU QUESTIONS WITH ANSWERS
Solution to Question Paper - June/July 2019

Module 1

1 (a) Explain the various features of OOC (8 Marks)

The various features of OOC 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 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 blueprint 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 in turn 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.

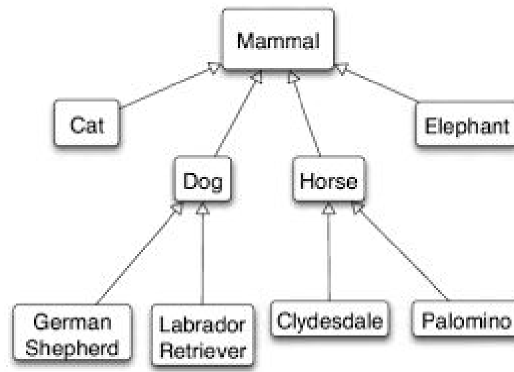
- Without inheritance, each object has to define all of its characteristics explicitly. But, by use of inheritance, an object needs to define only those qualities that make it unique within its class. It inherits its general attributes from its parent. Therefore, it is the inheritance mechanism that 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.

1 (b) What is a constructor? Mention its types. Explain the parameterized constructor with a suitable code. (8 Marks)

Constructor:

A **constructor** is a member function of a class which initializes objects of a class.
It appears as member function of each class whether it is defined or not.
It is automatically called when an object (instance of class) is created.
It has the same name as that of the class.
It may or may not take parameters.
It does not return anything, not even void.

The prototype of a constructor is

<class name> (<parameter list>);

Types of constructor:

Different types of constructors are
Zero argument or default constructor
Parameterized constructor
Explicit constructor
Copy constructors

PARAMETERIZED CONSTRUCTORS:

Parameterized constructors are constructors which takes one or more than one arguments.
The arguments help to initialize an object when it is created.
To create a parameterized constructor, we need to add parameters to the constructor.
Example program to illustrate parameterized constructor

```
// Example program to demonstrate parameterized constructor
#include <iostream>
using namespace std;

class construct
{
        public:
          int a,b;

          // Parameterized constructor
          construct(int x, int y)
          {
                    a = x;
                    b = y;
```

```
        }
};

int main()
{
        // Parameterized constructor called
        construct c(10,20);
        cout << "a = " << c.a << endl;
        cout << "b= " << c.b << endl;

        return 0;
}
```

Output:
```
$ g++ ParamConstructor.C
$ ./a.out
a = 10
b= 20
```

1 (c) Give the difference between procedure oriented programming and object oriented programming. (4 Marks)

ANS:
The table shows the difference between Procedure Oriented Programming and Object Oriented Programming.

#	**Procedure Oriented Programming**	**Object Oriented Programming**
1	Program is divided into small parts called functions	Program is divided into parts called objects
2	Focus is on procedures. The code is centered around procedures.	Focus is on data. Code is centered around data.
3	Procedures are dissociated from data and are not part of it.	Procedures are bound to the data.
4	Data is not secure. Compilers that implement the procedure-oriented programming system do not prevent unauthorized functions from accessing/manipulating structure variables.	Enables data security by throwing compile time errors against piece of code that violate the prohibition.
5	Data is not initialized	Provides guaranteed initialization of of data. Programmers can ensure a guaranteed initialization of data members of structure variables to the desired values.

6.	Overloading is not supported	Supports overloading of operators and functions
7	Top-down approach	Bottom-up approach
8	Doesn't support inheritance	Supports inheritance which allows one structure to inherit the characteristics of an existing structure.
9	Doesn't support polymorphism	Supports polymorphism which allows functions with different set of formal arguments to have the same name.
10	Doesn't Supports Encapsulation	Supports encapsulation, data and functions that act upon the data are enclosed within a single unit called class.

2 (a) What is an inline function? Write a C++ function to find the maximum of 2 numbers using inline. (8 Marks)

An **inline** function is a powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

Following is an example, which makes use of inline function to return max of two numbers −

```
#include <iostream>

using namespace std;

inline int Max(int x, int y) {
  return (x > y)? x : y;
}

// Main function for the program

int main() {

  cout << "Max (20,10): " << Max(20,10) << endl;
```

```
   cout << "Max (0,200): " << Max(0,200) << endl;

   cout << "Max (100,1010): " << Max(100,1010) << endl;


   return 0;

}
```

When the above code is compiled and executed, it produces the following result −

```
Max (20,10): 20

Max (0,200): 200

Max (100,1010): 1010
```

2 (b) Why friend function is required? Write a program to add two numbers using friend function. (8 Marks)

Friend function can be used as bridges between two classes.
To bridge two classes with a function, the function should be declared as a friend to both the classes.
Then the friend function can access private data of both classes.

C++ program to find the sum of two numbers using bridge friend function add()

```
#include <iostream>
using namespace std;

class B ;    // Forward declaration

//void add (A, B);

class A
{
        private:
                int a;

        public:
                A()
                {
                        a = 100;
                }
                friend void add(A,B);
};
```

```
class B
{
	private:
		int b;
	public:
		B()
		{
			b = 200;
		}
		friend void add(A,B);
};

void add (A Aobj, B Bobj)
{
	cout << "Sum of private members of class A and B = " <<  (Aobj.a + Bobj.b);
}

int main()
{
	A A1;
	B B1;
	add (A1, B1);
	return 0;
}
```

```
Output:
$ g++ bridge.C
$ ./a.out
Sum of private members of class A and B = 300
```

2 (c) Write short notes on function overloading.

Function Overloading:

- C++ allows two or more functions to have the same name, but they must have different signatures.
- **Signature of a function means the number, type, and sequence of formal arguments of the function.**
- In order to distinguish amongst the functions with the same name, the compiler expects their signature to be different.
- Depending upon the type of parameters that are passed to the function call, the compiler decides which of the available definitions will be invoked.
- For this, function prototypes should be provided to the compiler for matching the function calls.

- Accordingly the linker, during the link time, links the function call with the correct function definition.

Example program to demonstrate function overloading:

```
#include<iostream>
using namespace std;

int add(int,int);
int add (int,int,int);

int main()
{
	int x,y;
	x = add(10,20);
	y = add(30, 40, 50);
	cout << x << endl<< y << endl;
}

int add(int a, int b)
{
	return (a+b);
}

int add(int a, int b, int c)
{
	return (a+b+c);
}
```

Output:

```
$ g++ funcOverload.C
$ ./a.out
30
120
```

Module 2

3 (a) List and explain the Java Buzzwords (8 Marks)

The following is the list of Java buzzwords

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

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 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 behavior 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 has been designed with the goal of **"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.

3 (b) Describe the concept of bytecode (4 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).

Bytecodes are platform-independent instructions. So Java's bytecodes are portable, that is, the same bytecode can execute on any platform containing a JVM that understands the version of Java in which the bytecodes are compiled.
Bytecodes are executed by the Java Virtual Machine (JVM).

Advantages of Java or why Java is portable and more secure?
JVM was 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 run time package exists for a given system, any Java program can run on it. 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.

Bytecode files, such as Java CLASS files, are most often generated from source code using a compiler, like javac.

3 (c) Develop a program to calculate the average among the elements {4, 8, 10, 12 } using foreach in java. How foreach is different from for? (8 Marks)

```
// Progarm to calculate the average among the elements (4, 8, 10, 12} using foreach
public class JavaProgram
{
   public static void main(String args[])
   {

      int nums[] = { 4, 8, 10, 12};
      int sum = 0;

      // use the for-each style for loop to display and sum the values
      for(int x : nums)
      {
         sum = sum + x;
      }
```

```
        Float avg = sum/4;
        System.out.println("\nAverage =  " + avg);

    }
}
```

The following table gives the difference of for and foreach loop:

	For Loop	ForEach Loop
1	For Loop Variable Always int Only.	ForEach Loop Variable Same as Type of Values Under Array.
2	For loop Iterates a Statement or a Block of Statements Repeatedly until a Specified Expression Evaluates to False.	For-each loop is used to Iiterate through the Items in Object Collections, List Generic Collections or Array List Collections.
3	For Loops are Faster Than For Each Loop.	For Each Loop are Slower Than For Loop.

4	Need To Loop Bounds (Minimum,Maximum).
	
	Ex:
	int count=0;
	int j;
	for(int i=0;i<=5;i++)
	{
	j=count+1;
	}
		No Need To Loop Bounds Minimum or Maximum.
		
		EX:
		int z=0;
		int [] a=new int[] {0,1,2,3,4,5};
		foreach(int i in a)
		{
		z=z+1;
		}

4(a) List the different types of operators. Explain any three. (8 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 operands.
- In prefix form the operator precedes the operands.
- 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 the 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 the following two statements -
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;
    c = 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

Relational Operators:

- The relational operators determine the relationship between the two operands.
- They determine equality and ordering.
- The relational operators are

Sl. No.	Operator	Result
1	==	Equal to
2	!=	Not Equal to
3	>	Greater than
4	<	Less than
5	>=	Greater than or equal to
6	<=	Less than or equal to

- The outcome of these operations is a boolean value.

- The relational operators are used in the expressions that control the if statement and various loop statements.
- Any type in Java, including integers, floating-point numbers, characters and Booleans can be compared using the equality test, ==, and the inequality test, !=.
- only numeric types can be compared using the ordering operators. That is, only integer, floating-point, and character operands may be compared to see which is greater or less than the other.
- Example:

```
int a = 4;
int b = 1;
boolean c = a < b;
```

- The result of a < b (which is false) is stored in c.
- The C/C++ statements

```
int done;
if (!done)    ...
if (done)   ...
```

must be written like this

```
if ( done == 0)  ...
if ( done != 0)  ...
```

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

Logical Operators:

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

<table>
<tr><th>Sl. No.</th><th>Operator</th><th>Result</th></tr>
<tr><td>1</td><td>&</td><td>Logical AND</td></tr>
<tr><td>2</td><td>|</td><td>Logical OR</td></tr>
<tr><td>3</td><td>^</td><td>Logical XOR (Exclusive OR)</td></tr>
<tr><td>4</td><td>||</td><td>Short-circuit OR</td></tr>
<tr><td>5</td><td>&&</td><td>Short-circuit AND</td></tr>
<tr><td>6</td><td>!</td><td>Logical unary NOT</td></tr>
<tr><td>7</td><td>&=</td><td>AND assignment</td></tr>
<tr><td>8</td><td>|=</td><td>OR assignment</td></tr>
</table>

9	^=	XOR assignment
10	==	Equal to
11	!=	Not equal to
12	?:	Ternary if-then-else

The following table shows the effect of each logical operation:

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

```
// Program to demonstrate the boolean logical operators
class BoolLogic {
    public static void main(String args[ ]) {
        boolean a = true;
        boolean b = false;
        boolean c = a |  b;
        boolean d = a & b;
        boolean e = a ^ b;
        boolean f = (!a & b) | (a & !b);
        boolean g = !a;

        System.out.println("           a = " + a);
        System.out.println("           b = " + b);
        System.out.println("       a | b = " + c);
        System.out.println("       a & b = " + d);
        System.out.println("       a ^ b = " + e);
        System.out.println("!a&b | a& !b= " + f);
    }
}
```

Output:

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

Short-Circuit Logical Operators:

- Java provides two Boolean operators not found in many other computer languages. These are secondary versions of the Boolean AND and OR operators and are known as short-circuit operators.
- From the table we can see that, the OR operator results in true when A is true, no matter what B is. Similarly, the AND operator results in false when A is false, no matter what B is.
- **When we use || and && forms, rather than | and & forms of these operators, 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.

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

4(b) What is an array? List the types and explain any one with suitable code. (6 Marks)

What is an array?

- 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.

The types of arrays are

1. One Dimensional Array
2. Multi Dimensional Array

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 indices 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. ");
   }
}
```

4(c) Explain switch case with an example. (6 Marks)

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 values 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 to note:

- The **switch** differs from the **if** in that switch can only test for equality, whereas if can evaluate any type of Boolean expression. That is, 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.

Module 3

5(a) Explain the package in Java with an example (8 Marks)

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:

To create a package, 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.

This is 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.

A Short Package Example

```
// A simple package
package MyPack;
class Balance {
String name;
double bal;
Balance(String n, double b) {
name = n;
bal = b;
```

```
}
void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
class AccountBalance {
public static void main(String args[]) {
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
}
}
```

5(b) Explain the interfaces in Java (8 Marks)

Interfaces:

- Like a class, an interface can have methods and variables, but the methods declared in the interface are by default abstract (only method signature, no body)
- Interface specify what a class must do, but not how to do. It is the blueprint of the class.
- Once an interface is defined, any number of classes can implement.
- Also, one class can implement any number of interfaces.
- To implement an interface, a class must create the complete set of methods defined by the interface.
- By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.
- Interfaces are designed to support dynamic method resolution at run time.

Defining an Interface:

An interface is defined like a class. The general form of an interface is :

```
access interface name {
   return-type method-name1 (parameter-list);
   return-type method-name2 (parameter-list);
   type final-varname1 = value;
   type final-varname2 = value;
   //...
   return-type method-nameN(parameter-list);
   type final-varnameN = value;
}
```

access:

when no access specifier is included the interface is only available to all members of the package in which it is declared.
- When it is declared as public, the interface can be used by any other code.

name:
- name is the name of the interface and can be any valid identifier.

Method:
- The methods that are declared have no bodies, they end with a semicolon after the parameter list.
- They are abstract methods.
- There can be no default implementation of any method specified within an interface.
- Each class that included an interface must implement all of the methods.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then class must be declared **abstract**.

Variables:
- Variables can be declared inside interface declarations.
- They are implicitly public, final and static, meaning they cannot be changed by the implementation class.
- All methods and variables are implicitly public.
- Example:

```
interface Callback {
   void callback (int param);
}
```

Implementing Interfaces:
- once an interface has been defined, one or more classes can implement that interface.
- To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface.
- The general form of a class that includes the implements clause is -

```
class calssname [extends superclass] [implements interface [,interface...]] {
      // class-body
}
```

- If a class implements more than one interface, the interfaces are separated with a comma.
- If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface.
- **The methods that implement an interface must be declared public.**
- Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.
- Example:

```
class Client implements Callback {
   // Implement Callback's interface
   public void callback(int p)   {
```

```
        System.out.println(“callback called with “ + p);
    }
}
```

Note:
1. The method callback() is declared using the public access specifier.
2. When you implement an interface method, it must be declared as public.

- Classes that implement interfaces can define additional members of their own.

Example: here the class Client adds the method nonIfaceMethod()

```
class Client implements Callback {
   // Implement Callback’s interface
   public void callback(int p)  {
      System.out.println(“callback called with “ + p);
   }

    void nonIfaceMethod() {
        System.out.println(“Classes that implement interfaces “ +
                              “ may also define other members too. “);
    }
}
```

Accessing Implementations Through Interface References:

- We can declare variables as object references that use an interface.
- Any instance of any class that implements the declared interface can be referred to by such a variable.
- When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically
- at run time, allowing classes to be created later than the code which calls method on them.

Caution:

- Because dynamic lookup of a method at run time incurs a significant overhead when compared with the normal method invocation in Java, you should be careful not to use interfaces casually in performance-critical code.
- The following example calls the callback() method via an interface reference variable:

```
class TestIface  {
   public static void main(“String args[]) {
       Callback c = new Client ();
```

```
        c.callback(42);
    }
}
```

- Note that variable c is defined to be of the interface type Callback, yet it was assigned an instance of Client.
- **Although c can be used to access the callback() method, it cannot access any other members of the Client class.**
- An interface reference variable only has knowledge of the methods declared by its interface declaration. So c cannot be used to access **nonIfaceMethod**() since it is defined by Client and not Callback.

Example program to demonstrate the polymorphic power of an interface reference variable.

```
// defining the interface Callback
interface Callback  {
    void callback(int param);
}

// class Client implements the interface Callback
class Client implements Callback {
    // Implement Callback's interface
    public void callback(int p) {
        System.out.println("callback called with " + p);
    }
    void nonIfaceMethod() {
        System.out.println("Classes that implement interfaces " +
                                "may also define other members, too");
    }
}

// Another implementation of Callback
class AnotherClient implements Callback {
    // Implement Callback's interface
    public void callback(int p) {
        System.out.println("Another version of callback");
        System.out.println("p squared is " + (p*p));
   }
}

// Accessing implementation through interface references
class TestIface {
     public static void main(String args[]) {
        Callback c = new Client ();
        c.callback(2);           // c now refers to Client object
```

```
        AnotherClient ob = new AnotherClient();
        c = ob;  // c now refers to AnotherClient object
        c.callback(2);

        Client cliob = new Client ();
        cliob.nonIfaceMethod();
    }
}
```

The version of callback() that is called is determined by the type of object that c refers to at run time.

Output:

```
$ javac TestIface.java
$ java TestIface
callback called with 2
Another version of callback
p squared is 4
Classes that implement interfaces  may also define other members, too
```

5(c)Write short notes on 'this' keyword with an example (4 Marks)

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. this can be used inside any method to refer to the current object.

That is, **this** is always a reference to the object on which the method was invoked.

Box( ):

```
// A redundant use of this.
Box(double w, double h, double d) {
this.width = w;
this.height = h;
this.depth = d;
}
```

The use of **this** is redundant, but perfectly correct. Inside Box( ), this will always refer to the invoking object. It is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes. But we can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. However, when a local variable has the same name as an instance variable, the local variable hides the instance variable. This is why width, height, and depth were not used as the names of the parameters to the Box( ) constructor inside the Box class. If they had been, then width would have referred to the formal parameter, hiding the instance variable width.

While it is usually easier to simply use different names, there is another way around this situation. Because this lets you refer directly to the object, you can use it to resolve any namespace collisions that might occur between instance variables and local variables.

For example, here is another
version of Box( ), which uses width, height, and depth for parameter names and then uses this to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;
}
```

6(a)Explain exception handling with a suitable code (8 Marks)

General Definition of Exception:
An exception is an abnormal condition that arises in a code sequence at run time. An exception is a run-time error.

In computer languages that do not support exception handling, errors must be checked and handled manually through the use of error codes. This approach is cumbersome as it is troublesome.

Java's exception handling avoids these problems and brings run-time error management into the object-oriented world.o

Exception Handling Fundamentals:

- A Java **exception** is an object that describes an exceptional (error) condition that has occurred in a 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 support some error condition to the caller of a method.

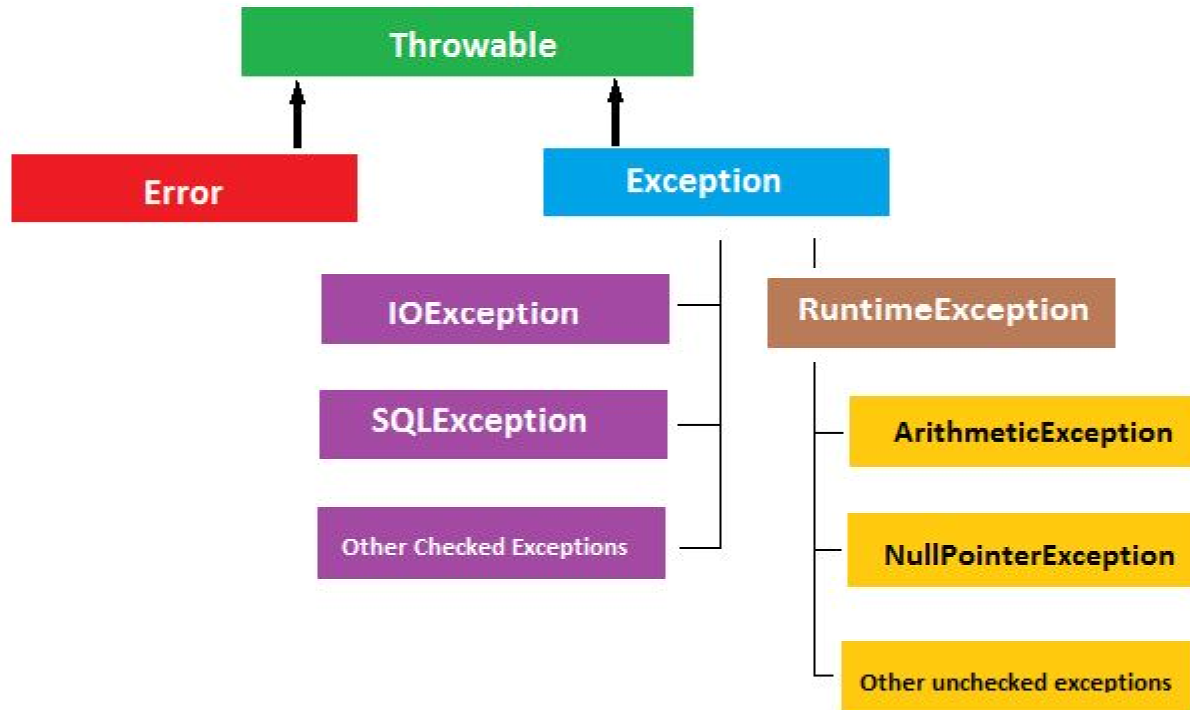
- Java exception handling is managed via five keywords:
  - try
  - catch,
  - throw
  - throws
  - finally
- Program statements that you want to monitor for exceptions are contained within a **try** block.
- If an exception occurs within the **try** block, it is thrown.
- Your code can catch this exception, using **catch**, and handle it.
- System generated exceptions are automatically thrown by the Java run-time system.
- To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by **throws** clause.
- Any code that absolutely must be executed after the try block completes is put in a **finally** block
- The general form of an exception-handling block is shown below:

```
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
}
```

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.



The following program includes try block and a catch clause.

```
class Exc2 {
        public static void main(String args[]) {
        int d,a;
        try {
                d = 0;
                a = 42 / d;
                System.out.println("This will not be printed");
        }
```

```
            catch (ArithmeticException e) {
                System.out.println("Division by zero");
            }
            System.out.println("After catch statement");
           }
    }
```

Output:
Division by zero
After catch statement

6(b) Explain the Java Garbage Collection (8 Marks)

In Java objects are dynamically allocated by using the new operator, it handles deallocation automatically. The technique that accomplishes this is called **garbage collection**.

When no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. Garbage collection only occurs sporadically (if at all) during the execution of your program.

It will not occur simply because one or more objects exist that are no longer used.

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resources 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.

6(c) Write short notes on “super” keyword, with an example (4 Marks)

super is the keyword using which the subclass refers to its immediate superclass.
super has two general forms
1. The first calls the superclass’ constructor.
2. The second is used to access a member of the superclass that has been hidden by member of a subclass.
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.

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 the use 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
```

Module 4

7(a) Explain the concepts of multithreading in Java. Explain the two ways of making class threadable with examples. (10 Marks)

A **thread** is a path of execution within a process. A process can contain multiple threads.

Java provides built-in support for multi threaded programming. A multi-threaded program contains two or more parts that can run concurrently. Each part of such a program is called a **thread**, and each thread defines a separate path of execution. Thus multi threading is a specialized form of multitasking.

Creating a Thread

A thread can be created in two ways

1. we can implement the **Runnable** interface

2. we can extend the **Thread** class

1. Creating a thread by Implementing Runnable interface:

To create a thread by implementing Runnable interface, create a class that implements the **Runnable** interface.

Runnable abstracts a unit of executable code.

To implement Runnable, a class has to implement a single method called run(), which is declared like this -

public void run()

Inside run(), you will define the code that constitutes the new thread.

Run() can call other methods, use other classes and declare variables just like the main thread can.

The only difference is that run() establishes the entry point for another concurrent thread of execution within your program. This thread will end when run() returns.

After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. One such constructor is

Thread (Runnable threadOb, String threadName)

In this constructor, threadOb is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by threadName.

After the new thread is created, it will not start running until you call its start() method, which is declared within Thread. In essence, start executes a call to run().

The start() method is shown here

void start()

Example program that creates a new thread and starts it running:

```
class NewThread implements Runnable  {
        public void run()  {
                        System.out.println("Thread is running");
        }
}

// Main class
public class MThreadImp  {
        public static void main(String args[])   {
                NewThread threadob = new NewThread();
                Thread obj = new Thread(threadob);
                obj.start();
        }
}
```

Output:
```
$ javac MThreadImp.java
$ java MThreadImp
Thread is running
```

Note:

In a multi-threaded program, often the main thread must be the last thread to finish running.

In older JVMs, if the main thread finishes before a child thread has completed, then the Java run-time system may "hang".

The above program ensures that the main thread finished last, because the main thread sleeps for 1000 milliseconds between iterations, but the child sleeps for only 500 milliseconds.

This causes the child thread to terminate earlier than the main thread.

2. Creating a thread by extending Thread class:

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.

The extending class must override the run() method, which is the entry point for the new thread.

It must also call start() to begin execution of the new thread.

The following is the example program -

```
// Create a second thread by extending Thread
class NewThread extends Thread {
	public void run() {
			System.out.println("Thread is running");
	}
}

// Main class
public class MThread {
	public static void main(String args[]) {
		NewThread threadob = new NewThread();
		threadob.start();
	}
}
```

Output:
```
$ javac MThread.java
$ java MThread
Thread is running
```

7(b) With a syntax, explain isAlive() and join() with suitable program. (10 Marks)

Using isAlive() and join():

In a multi-threaded program, often the main thread must be the last thread to finish running.
This is accomplished by calling sleep() within main(), with a long enough delay to ensure that all child threads terminate prior to the main thread.
But, this is not the right method as the child might take more than the sleep time.
If main thread comes to know if all child threads has ended or not then, problem is solved.
The solution to this problem is there should be a way to know if the thread has ended or not.
There are two ways to determine whether a thread has finished.

1. isAlive() method:

- First, you can call **isAlive**() on the thread. Its general form is
  final boolean isAlive()
- The isAlive method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.

2. join() method:

- The second method that will be more commonly used to wait for a thread to finish is called **join**(). Its general form is
  final void join() throws InterruptedException
- This method waits until the tread on which it is called terminates.
- **Its name comes from the concept of the called thread waiting until the specified thread joins.**
- Additional forms of join() allows to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Example program to demonstrate the use of join() and isAlive():

```
// using join() to wait for threads to finish.
class NewThread extends Thread  {
        String name;

        NewThread (String threadname)  {
                name = threadname;
                System.out.println("New Thread: " + name);
        }
```

```
        public void run()  {
                try  {
                        for(int i = 5 ; i >0 ; i--)  {
                                System.out.println( name + " :  "  + i);
                        Thread.sleep(1000);
                }
         }
         catch( InterruptedException e)  {
                System.out.println("Caught : " + e);
         }
       }

}

// Main class
class DemoNewJoin   {
   public static void main(String args[])   {
      NewThread ob1 = new NewThread ("One");
      NewThread ob2 = new NewThread ("Two");
      NewThread ob3 = new NewThread ("Three");

      ob1.start();
      ob2.start();
      ob3.start();

        System.out.println("Thread one is alive: " + ob1.isAlive());
        System.out.println("Thread Two is alive: " + ob2.isAlive());
        System.out.println("Thread Three is alive: " + ob3.isAlive());

        // wait for threads to finish
        try   {
           System.out.println("Waiting for threads to finish.");
           ob1.join();
           ob2.join();
           ob3.join();
        }
      catch (InterruptedException e) {
         System.out.println("Main thread Interrupted.");
      }
      System.out.println("Thread one is alive: " + ob1.isAlive());
        System.out.println("Thread Two is alive: " + ob2.isAlive());
```

```
        System.out.println("Thread Three is alive: " + ob3.isAlive());

      System.out.println("Main thread exiting.");
   }
}
```

Output:

```
$ javac DemoNewJoin.java
$ java DemoNewJoin
New Thread: One
New Thread: Two
New Thread: Three
One :  5
Two :  5
Thread one is alive: true
Three :  5
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One :  4
Two :  4
Three :  4
One :  3
Two :  3
Three :  3
One :  2
Two :  2
Three :  2
One :  1
Two :  1
Three :  1
Thread one is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

Note:

You can observe that, after the calls to join() return, the threads have stopped executing.

8(a) Write short notes on Event Listener and explain any two interfaces with syntax (8 Marks)

Event Listeners are created by implementing one or more of the interfaces defined by the java.awt.event package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument. The following table shows commonly used listener interfaces and provides a brief description of the methods that they define.

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

The AdjustmentListener Interface
This interface defines the adjustmentValueChanged( ) method that is invoked when an adjustment event occurs. Its general form is shown here:
void adjustmentValueChanged(AdjustmentEvent ae)

The ComponentListener Interface
This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:
void componentResized(ComponentEvent ce)
void componentMoved(ComponentEvent ce)
void componentShown(ComponentEvent ce)
void componentHidden(ComponentEvent ce)

8(b) Write short notes on Event class and explain any two with syntax (8 Marks)

The classes that represent events are at the core of Java's event handling mechanism. Java defines several types of events. The most widely used events are those defined by the AWT and those defined by Swing. The table shows the main event classes.

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

The ActionEvent Class

An ActionEvent is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. The ActionEvent class defines four integer constants that can be used to identify any modifiers associated with an action event: ALT_MASK, CTRL_MASK, META_MASK, and SHIFT_MASK. In addition, there is an integer constant, ACTION_ PERFORMED, which can be used to identify action events.

ActionEvent has these three constructors:
ActionEvent(Object src, int type, String cmd)
ActionEvent(Object src, int type, String cmd, int modifiers)
ActionEvent(Object src, int type, String cmd, long when, int modifiers)

Here, src is a reference to the object that generated this event. The type of the event is specified by type, and its command string is cmd. The argument modifiers indicates which modifier keys (ALT , CTRL , META , and/or SHIFT) were pressed when the event was generated. The when parameter specifies when the event occurred.

We can obtain the command name for the invoking ActionEvent object by using the

getActionCommand( ) method, shown here:
String getActionCommand( )

For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.
The getModifiers( ) method returns a value that indicates which modifier keys (ALT , CTRL , META , and/or SHIFT) were pressed when the event was generated. Its form is shown here:
int getModifiers( )

The method getWhen( ) returns the time at which the event took place. This is called the event's timestamp. The getWhen( ) method is shown here:
long getWhen( )

The TextEvent Class
Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program. TextEvent defines the integer constant TEXT_VALUE_CHANGED.
The one constructor for this class is shown here:
TextEvent(Object src, int type)
Here, src is a reference to the object that generated this event. The type of the event is specified by type.
The TextEvent object does not include the characters currently in the text component that generated the event. Instead, your program must use other methods associated with the text component to retrieve that information. This operation differs from other event objects discussed in this section. For this reason, no methods are discussed here for the TextEvent class. Think of a text event notification as a signal to a listener that it should retrieve information from a specific text component.

8 (c) How inner classes are used in Java? Explain (4 Marks)

INNER CLASSES:

- **Inner class is a class defined within another class or even within an expression.**
- Inner classes can be used to simplify the code when using event adapter classes.
- Example program to illustrate inner class

Program Explanation:

- The goal of the applet is to display the string "Mouse Pressed" in the status bar of the applet viewer or browser when the mouse is pressed.
- InnerClassDemo is a top level class that extends Applet
- MyMouseAdapter is an inner class that extends MouseAdapter.

- As, MyMouseAdapter is defined within the scope of InnerClassDemo, it has access to all the variables and methods within the scope of that class.
- Therefore, the mousePressed() method can call the showStatus() method directly.
- It no longer needs to do this via a stored reference to the applet.
- It is no longer necessary to pass MyMouseAdapter() a reference to the invoking object.

```
// inner class demo import java.awt.event.*; // Contains all Event
Listener Interfaces import java.applet.*; // For Applets

/* <applet code = "InnerClassDemo" width=500 height=300>
</applet> */ public class InnerClassDemo extends Applet {
    public void init() {
   addMouseListener(new MyMouseAdapter()); } class MyMouseAdapter
    extends MouseAdapter { // class within a class
      public void mousePressed(MouseEvent me) {
          showStatus("Mouse Pressed"); } } }
```

Module 5

9(a) What is an applet? Explain the life cycle of an applet. (10 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().

9(b) Explain passing parameters in Applets (10 Marks)

PASSING PARAMETERS TO APPLETS:

- The APPLET tag in HTML allows the user to pass parameters to your applet.
- To retrieve a parameter, use the getParameter() method. It returns the value of the specified parameter in the form of a String object.
- Thus, for numeric and boolean values, we have to convert their string representations into their internal formats.
- Conversions to numeric types must be attempted in a try statement that catches NumberFormatException. Uncaught exceptions should never occur within an applet.
- We should test the return values from getParameter(). If a parameter is not available, getParameter() will return null.
- Example to demonstrate passing parameters

```
// Use Parameters
import java.awt.*;
import java.applet.*;

/*
<applet code="ParamDemo" width=500 height=300>
<param name = fontName value=TimesNewRoman>
<param name = fontSize  value=20>
<param name = leading value=2>
<param name = accountEnabled  value=true>
</applet>
*/

public class ParamDemo extends Applet {
        String fontName;
        int fontSize;
        float leading;
        boolean active;

        //Initialize the string to be displayed
        public void start()   {
                String param;

                fontName = getParameter("fontName");
                if (fontName == null)
                        fontName = "Not Found";

                param = getParameter("fontSize");
                try  {
                        if (param != null)   // if not found
                                fontSize = Integer.parseInt(param);
                        else
```

```
                    fontSize = 0;
        }
        catch (NumberFormatException  e)  {
                fontSize = -1;
        }

        param = getParameter("leading");
        try  {
                if (param != null)   // if not found
                        leading = Float.valueOf(param).floatValue();
                else
                        leading = 0;
        }
        catch (NumberFormatException  e)  {
                leading = -1;
        }

        param = getParameter ("accountEnabled");
        if (param != null)
                active = Boolean.valueOf(param).booleanValue();
        }

        // Display parameters
        public void paint(Graphics g)  {
                g.drawString("Font name: " + fontName, 0, 10);
                g.drawString("Font Size :" + fontSize, 0, 26);
                g.drawString("Leading : " + leading, 0, 42);
                g.drawString("Account Active :" + active, 0, 58);
        }
}
```

10 (a) Explain the following with a suitable code (20 Marks)

i) JLabel

JLabel is Swing's easiest-to-use component. JLabel can be used to display text and/or an icon. It is a passive component in that it does not respond to user input. JLabel defines several constructors.

Here are three of them:

JLabel(Icon icon)

JLabel(String str)

JLabel(String str, Icon icon, int align)

Here, str and icon are the text and icon used for the label. The align argument specifies the horizontal alignment of the text and/or icon within the dimensions of the label. It must be

one of the following values: LEFT, RIGHT, CENTER, LEADING, or TRAILING. These constants are defined in the SwingConstants interface, along with several others used by the Swing classes.
Icons are specified by objects of type Icon, which is an interface defined by Swing. The easiest way to obtain an icon is to use the ImageIcon class. ImageIcon
implements Icon and encapsulates an image. Thus, an object of type ImageIcon can be passed as an argument to the Icon parameter of JLabel's constructor. There are several ways to provide the image, including reading it from a file or downloading it from a URL. Here is the ImageIcon constructor:

ImageIcon(String filename)
It obtains the image in the file named filename.
The icon and text associated with the label can be obtained by the following methods:
Icon getIcon( )
String getText( )
The icon and text associated with a label can be set by these methods:
void setIcon(Icon icon)
void setText(String str)
Here, icon and str are the icon and text, respectively. Therefore, using setText( ) it is possible to change the text inside a label during program execution.
The following applet illustrates how to create and display a label containing both an icon and a string. It begins by creating an ImageIcon object for the file france.gif, which depicts the flag for France. This is used as the second argument to the JLabel constructor. The first and last arguments for the JLabel constructor are the label text and the alignment. Finally, the label is added to the content pane.

```
// Demonstrate JLabel and ImageIcon.
import java.awt.*;
import javax.swing.*;
/*
<applet code="JLabelDemo" width=250 height=150>
</applet>
*/
public class JLabelDemo extends JApplet {
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() {
// Create an icon.
ImageIcon ii = new ImageIcon("france.gif");
// Create a label.
JLabel jl = new JLabel("France", ii, JLabel.CENTER);
// Add the label to the content pane.
add(jl);
}
}
```

ii) JTextField

JTextField is the simplest Swing text component. It is also probably its most widely used text component. JTextField allows you to edit one line of text. It is derived from JTextComponent, which provides the basic functionality common to Swing text components. JTextField uses the Document interface for its model.

Three of JTextField's constructors are shown here:

```
JTextField(int cols)
JTextField(String str, int cols)
JTextField(String str)
```

Here, str is the string to be initially presented, and cols is the number of columns in the text field. If no string is specified, the text field is initially empty. If the number of columns is not specified, the text field is sized to fit the specified string.
JTextField generates events in response to user interaction. For example, an ActionEvent is fired when the user presses ENTER . A CaretEvent is fired each time the caret (i.e., the cursor) changes position. (CaretEvent is packaged in javax.swing.event.) Other events are also possible. In many cases, your program will not need to handle these events. Instead, you will simply obtain the string currently in the text field when it is needed. To obtain the text currently in the text field, call getText( ).
The following example illustrates JTextField. It creates a JTextField and adds it to the content pane. When the user presses ENTER , an action event is generated. This is handled by displaying the text in the status window.

```
// Demonstrate JTextField.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JTextFieldDemo" width=300 height=50>
</applet>
*/
public class JTextFieldDemo extends JApplet {
JTextField jtf;
```

```
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 text field to content pane.
jtf = new JTextField(15);
add(jtf);
jtf.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent ae) {
// Show text when user presses ENTER.
showStatus(jtf.getText());
}
});
}
}
```

iii) JList

In Swing, the basic list class is called JList. It supports the selection of one or more items from a list. Although the list often consists of strings, it is possible to create a list of just about any object that can be displayed. JList is so widely used in Java that it is highly unlikely that you have not seen one before.

JList provides several constructors. The one used here is
JList(Object[ ] items)

This creates a JList that contains the items in the array specified by items.
JList is based on two models. The first is ListModel. This interface defines how access to the list data is achieved. The second model is the ListSelectionModel interface, which defines methods that determine what list item or items are selected.
Although a JList will work properly by itself, most of the time you will wrap a JList inside a JScrollPane. This way, long lists will automatically be scrollable, which simplifies GUI design. It also makes it easy to change the number of entries in a list without having to change the size of the JList component.

A JList generates a ListSelectionEvent when the user makes or changes a selection.
This event is also generated when the user deselects an item. It is handled by implementing ListSelectionListener. This listener specifies only one method, called valueChanged( ), which is shown here:

```
void valueChanged(ListSelectionEvent le)
```

Here, le is a reference to the object that generated the event. Although ListSelectionEvent does provide some methods of its own, normally you will interrogate the JList object itself to determine what has occurred. Both ListSelectionEvent and ListSelectionListener are packaged in javax.swing.event.

By default, a JList allows the user to select multiple ranges of items within the list, but you can change this behavior by calling setSelectionMode( ), which is defined by JList. It is shown here:

```
void setSelectionMode(int mode)
```

Here, mode specifies the selection mode. It must be one of these values defined by ListSelectionModel:

SINGLE_SELECTION
SINGLE_INTERVAL_SELECTION
MULTIPLE_INTERVAL_SELECTION

The default, multiple-interval selection, lets the user select multiple ranges of items within a list. With single-interval selection, the user can select one range of items. With single selection, the user can select only a single item. Of course, a single item can be selected in the other two modes, too. It's just that they also allow a range to be selected.

You can obtain the index of the first item selected, which will also be the index of the only selected item when using single-selection mode, by calling getSelectedIndex( ), shown here:

```
int getSelectedIndex( )
```

Indexing begins at zero. So, if the first item is selected, this method will return 0. If no item is selected, –1 is returned.

Instead of obtaining the index of a selection, you can obtain the value associated with the selection by calling getSelectedValue( ):

```
Object getSelectedValue( )
```

It returns a reference to the first selected value. If no value has been selected, it returns null.

The following applet demonstrates a simple JList, which holds a list of cities. Each time a city is selected in the list, a ListSelectionEvent is generated, which is handled by the valueChanged( ) method defined by ListSelectionListener. It responds by obtaining the index of the selected item and displaying the name of the selected city in a label.

```
// Demonstrate JList.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code="JListDemo" width=200 height=120>
</applet>
*/
public class JListDemo extends JApplet {
```

```
JList jlst;
JLabel jlab;
JScrollPane jscrlp;
// Create an array of cities.
String Cities[] = { "New York", "Chicago", "Houston",
"Denver", "Los Angeles", "Seattle",
"London", "Paris", "New Delhi",
"Hong Kong", "Tokyo", "Sydney" };
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 JList.
jlst = new JList(Cities);
// Set the list selection mode to single selection.
jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
// Add the list to a scroll pane.
jscrlp = new JScrollPane(jlst);
// Set the preferred size of the scroll pane.
jscrlp.setPreferredSize(new Dimension(120, 90));
// Make a label that displays the selection.
jlab = new JLabel("Choose a City");
// Add selection listener for the list.
jlst.addListSelectionListener(new ListSelectionListener() {
public void valueChanged(ListSelectionEvent le) {
// Get the index of the changed item.
int idx = jlst.getSelectedIndex();
// Display selection, if item was selected.
if(idx != -1)
jlab.setText("Current selection: " + Cities[idx]);
else // Otherwise, reprompt.
jlab.setText("Choose a City");
}
```

```
});
// Add the list and label to the content pane.
add(jscrlp);
add(jlab);
}
}
```

iv) JTable

JTable is a component that displays rows and columns of data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position. Depending on its configuration, it is also possible to select a row, column, or cell within the table, and to change the data within a cell. JTable is a sophisticated component that offers many more options and features than can be discussed here. (It is perhaps Swing's most complicated component.) However, in its default configuration, JTable still offers substantial functionality that is easy to use—especially if you simply want to use the table to present data in a tabular format. The brief overview presented here will give you a general understanding of this powerful component.

Like JTree, JTable has many classes and interfaces associated with it. These are packaged in javax.swing.table.

At its core, JTable is conceptually simple. It is a component that consists of one or more columns of information. At the top of each column is a heading. In addition to describing the data in a column, the heading also provides the mechanism by which the user can change the size of a column or change the location of a column within the table. JTable does not provide any scrolling capabilities of its own. Instead, you will normally wrap a JTable inside a JScrollPane.

JTable supplies several constructors. The one used here is

JTable(Object data[ ][ ], Object colHeads[ ])

Here, data is a two-dimensional array of the information to be presented, and colHeads is a one-dimensional array with the column headings.

JTable relies on three models. The first is the table model, which is defined by the TableModel interface. This model defines those things related to displaying data in a two-dimensional format. The second is the table column model, which is represented by TableColumnModel. JTable is defined in terms of columns, and it is TableColumnModel that specifies the characteristics of a column. These two models are packaged in javax.swing.table. The third model determines how items are selected, and it is specified by the ListSelectionModel, which was described when JList was discussed.

A JTable can generate several different events. The two most fundamental to a table's operation are ListSelectionEvent and TableModelEvent. A ListSelectionEvent is generated when the user selects something in the table. By default, JTable allows you to select one or more complete rows, but you can change this behavior to allow one or more columns, or one or more individual cells to be selected. A TableModelEvent is fired when that table's data changes in some way. Handling these events requires a bit more work than it does to handle the events generated by the previously described components and is beyond the scope of this book. However, if you simply want to use JTable to display data (as the following example does), then you don't need to handle any events.

Here are the steps required to set up a simple JTable that can be used to display data:

1. Create an instance of JTable.
2. Create a JScrollPane object, specifying the table as the object to scroll.
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane.

The following example illustrates how to create and use a simple table. A one-dimensional array of strings called colHeads is created for the column headings. A two-dimensional array of strings called data is created for the table cells. You can see that each element in the array is an array of three strings. These arrays are passed to the JTable constructor. The table is added to a scroll pane, and then the scroll pane is added to the content pane. The table displays the data in the data array. The default table configuration also allows the contents of a cell to be edited. Changes affect the underlying array, which is data in this case.

```
// Demonstrate JTable.
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTableDemo" width=400 height=200>
</applet>
*/
public class JTableDemo extends JApplet {
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() {
// Initialize column headings.
String[] colHeads = { "Name", "Extension", "ID#" };
// Initialize data.
Object[][] data = {
{ "Gail", "4567", "865" },
{ "Ken", "7566", "555" },
{ "Viviane", "5634", "587" },
{ "Melanie", "7345", "922" },
{ "Anne", "1237", "333" },
```

```
{ "John", "5656", "314" },
{ "Matt", "5672", "217" },
{ "Claire", "6741", "444" },
{ "Erwin", "9023", "519" },
{ "Ellen", "1134", "532" },
{ "Jennifer", "5689", "112" },
{ "Ed", "9030", "133" },
{ "Helen", "6751", "145" }
};
// Create the table.
JTable table = new JTable(data, colHeads);
// Add the table to a scroll pane.
JScrollPane jsp = new JScrollPane(table);
// Add the scroll pane to the content pane.
add(jsp);
}
}
```