PERIYAR UNIVERSITY

(NAAC 'A++' Grade with CGPA 3.61 (Cycle - 3)
State University - NIRF Rank 56 - State Public University Rank 25
SALEM - 636 011

CENTRE FOR DISTANCE AND ONLINE EDUCATION

(CDOE)

BACHELOR OF COMPUTER SCIENCE

SEMESTER - IV

JAVA PROGRAMMING

(Candidates admitted from 2024 onwards)

PERIYAR UNIVERSITY

CENTRE FOR DISTANCE AND ONLINE EDUCATION (CDOE)

B.Sc. COMPUTER SCIENCE 2024 admission onwards

Core Course VII
Java Programming

Prepared by:

Centre for Distance and Online Education (CDOE)
Periyar University, Salem – 11.

LIST OF CONTENTS

UNIT I

Objective: To provide fundamental knowledge of object-oriented programming and basics of Java.

REVIEW OF OBJECT ORIENTED CONCEPTS

Object

Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory.

Class

Collection of objects is called class. It is a logical entity.

A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance

When one object acquires all the properties and behaviours of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

If one task is performed in different ways, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.

In Java, we use method overloading and method overriding to achieve polymorphism.

Encapsulation

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

HISTORY OF JAVA

Java History

- Java is a high-level programming language originally developed by Sun Microsystems and released in 1991.
- Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX. So java is platform independent.
- Java was designed for the development of software for consumer electronic devices like TVs, VCRs, toasters and such other electronic devices.
- 1990 → A team of Sun Microsystems headed by James Gosling was decided to develop a special software that can be used to manipulate consumer electronic devices.
- 1991 → The team announced a new language named “oak”.
- 1992 → The team demonstrated the application of their new language to control a list of home applications.
- 1993 → The team known as Green Project team came up with the idea of developing web applets.
- 1994 → The team developed a web browser called “HotJava” to locate and run applet programs on internet.
- 1995 → Oak was renamed Java.
- 1996 → Sun releases Java Development Kit 1.0.
- 1997→ Sun releases Java Development Kit 1.1.
- 1998 → Sun releases the Java 2 with version 1.2.
- 1999 → Sun releases standard edition (J2SE) and enterprise edition(J2EE).
- 2000 → J2SE with SDK(software development kit) 1.3 was released.
- 2002 → J2SE with SDK 1.4 was released.
- 2004 → J2SE JDK 5.0 was released. This is known as J2SE 5.0.

JAVA BUZZWORDS

Object Oriented:

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

Robust and Secure:

- Java is a robust language. It provides many safeguards to ensure reliable code. It has strict compile time and runtime checking for data types.
- It is designed as garbage collected language relieving the programmers virtually all memory management problems.
- Java also incorporates the concept of exception handling which captures series errors and eliminates risk of crashing the system.
- The absence of pointers in java ensures that programs cannot gain access to memory locations without proper authorization.

Distributed:

- Java is designed as a distributed language for creating applications on networks. It has the ability to share both data and programs.
- Java applications can open and access remote objects on internet as easily as they can do in a local system.
- This enables multiple programmers at multiple remote locations to collaborate and work together on a single project.

Simple, Small and Familiar:

- Java is a small and simple language. Many features of C and C++ that are either redundant

  For example java does not use pointers, preprocessor header files, goto statement and overloading and multiple inheritance and many others.

Multithreaded and Interactive:

- Multithreaded means handling multiple tasks simultaneously. Java supports multithreaded programs. This means that we need not wait for the application to finish one task before beginning another.

  For example we can listen to an audio clip while scrolling a page and at the same time download an applet from a distant computer.
- This feature greatly improves the interactive performance of graphical applications.

High performance

- Java performance is impressive for an interpreted language, mainly due to the use of intermediate byte code. According to sun, java speed is comparable to the native C/C++.
- Java architecture is also designed to reduce overheads during runtime. Further, the incorporation of multithreading enhances the overall execution speed of java programs.

Dynamic and Extensible

- Java is a dynamic language.
- It is capable of dynamically linking in new class libraries, methods, and objects.
- Java program support functions written in other languages such as C and C++. These functions are known as **native methods.**

JVM ARCHITECTURE

It facilitates the execution of programs developed in java. It comprises of the following:

- **Java Virtual machine(JVM):** It is a program that interprets the intermediate java byte code and generates the desired output. It is because if byte code and JVM concepts that programs written in Java are highly portable.

- **Runtime class libraries:** There are a set of core class libraries that are required for the execution of java programs.``
- **User interface toolkits:** AWT and swing are examples of toolkits that support varied input methods for the users to interact with application program.
- **Deployment technologies:** JRE comprises the following key deployment technologies:
- **Java plug-in:** Enables the execution of a java applet on the browser.
- **Java Web start:** Enables remote-deployment of an application.

DATA TYPES

Data types specify the size and type of values that can be stored.

Data types in Java

Integer Types

✓ Integer types can hold whole numbers such as 123, -96, 5678. Java supports four types of integers. They are **byte, short, int, and long.**

Type	Size	Minimum value	Maximum value
Byte	One byte	-128	127
Short	Two bytes	-32,768	32,767
Int	Four bytes	-2,147,483,648	2,147,483,647
Long	Eight bytes	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

Numeric

Floating Point Types

✓ Floating point type contains fractional parts such as 26.78 and -7.890.

✓ The **float** type values are single-precision numbers while the **double** types represent double-precision numbers.

✓ Floating point numbers are treated as double-precision quantities. We must append f or F to the numbers. Example: 1.23f 7.67567e5F

Double-precision types are used when we need greater precision in storage of floating point numbers. Floating point data types support a special value known as Not-a-Number (NaN).

It is used to represent the result of operations such as dividing by zero, where an actual number is not produced.

Type	Size	Minimum value	Maximum value
Float	4 bytes	3.4e-038	1.7e+0.38
double	8 bytes	3.4e-038	1.7e+308

Character Type

✓ Java provides a character data type called **char**.

✓ The char type assumes a size of 2 bytes but, basically, it can hold only a single character.

Boolean Type

✓ It is used to test a particular condition during the execution of the program.

✓ There are only two values that a boolean type can take: true or false.

✓ Boolean type is denoted by the keyword boolean and uses only one bit of storage.

VARIABLES

A variable is an identifier that denotes a storage location used to store a data value. Variable names may consist of alphabets, digits, the underscore(_) and dollar characters, subject to the following conditions:

- They must not begin with a digit.
- Uppercase and lowercase are distinct.
- It should not be a keyword.
- White space is not allowed.
- Variable names can be of any length.

A variable must be given a value after it has been declared it is used in an expression. This can be achieved in two ways:

1. By using an assignment statement

2. By using a read statement

SCOPE AND LIFE TIME OF VARIABLE

Java variables are actually classified into three types:

- Instance variables
- Class variables
- Local variables
- Instance and class variable are declared inside a class. Instance variables are created when the objects are instantiated and they are associated with the objects.
- Class variables are global to a class and belong to the entire set of objects that class creates. Only one memory location is created for each class variable.
- Variables declared and used inside methods are called local variables. They are not available for use outside method definition.

We often encounter situations where there is a need to store a value on one type into a variable of another type.

✓ In such situation, we must cast the value to be stored by proceeding it with the type name in parentheses. The syntax is

type variable1 = (type) variable2;

The process of converting one data type to another is called **casting.** Examples:

int m= 50;

byte n = (byte)m;

Four integer types can be cast to any other type except Boolean. Casting into a smaller type may result in loss of data. Similarly, the float and double can be cast to any other type except Boolean.

From	To
byte	short, char, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double

long	float, double
float	Double

Casts that results in no loss of information

ARRAYS

An array is a group of related data items that share a common name. A particular value is indicated by specifying a number called index number or subscript in square brackets after the array name.

The arrays can be classified into two types. They are

1. One-dimensional array
2. Two-dimensional array

Creating An Array

Creation of array includes three steps:

1. Declare the array
2. Create memory locations
3. Put values into the memory locations.

One-dimensional array

A list of items can be given one variable name using only one subscript and such variable is called a single-subscripted variable or a one-dimensional array.

Declaration of one-dimensional array:

Arrays in java may be declared in two forms.

Form 1: type arrayname[ ];

Form 2: type[ ] arrayname;

Ex: int mark[];
int[ ] mark;

Creation of one-dimensional array

Arrays are created by using new operator. The general form is

arrayname = new type[size];

Ex: mark = new int[10];

Initialization of one-dimensional array:

Values are assigned to the array by specifying the subscript.

arrayname[subscript] = value;

Ex: mark[2] = 65;

An array may also be initialized when they are declared.

type arrayname[ ] = {list-of-values};

The list-of-values are separated by comma and surrounded by curly braces. The memory for the array is allocated by the compiler based on the number of values given.

Ex: int x[] = {25,35,15,5,55}

Array length : In java, all arrays store the allocated size in a variable named length. To know the size of an array, it can be accessed as

arrayname.length

Two-Dimensional Arrays

To store the values in a table form then two dimensional array is used. Two subscript are needed to access a value in two dimensional array.

Declaration of two-dimensional array:

Arrays in java may be declared in two forms.

Form 1: type arrayname[ ][ ];

Form 2: type[ ][ ] arrayname;

Ex: int mata[ ][ ];

int[ ][ ] mata;

Creation of two-dimensional array

Arrays are created by using new operator. The general form is

arrayname = new type[row][col];

Ex: mata = new int[3][2];

Initialization of two-dimensional array:

Values are assigned to the array by specifying the subscript.

arrayname[subscript1][subsrcipt2] = value;

Ex: mata[2][2] = 15;

An array may also be initialized when they are declared.

type arrayname[ ][ ] = {list-of-values};

The list-of-values are separated by comma and surrounded by curly braces. The memory for the array is allocated by the compiler based on the number of values given.

Ex: int x[ ][ ] = {25,35,15,5,55,45}

Variable Size Arrays

Java treats multidimensional arrays as “arrays of arrays”. It is possible to declare a two-dimensional array as follows.

int x[ ][ ] = new int [3][ ];

x[0] = new int[2];

x[1] = new int[5];

x[2] = new int[3];

OPERATORS

- Java supports a rich set of operators.
- An operator is a symbol that is used for manipulate data and variables.
- Operators are used in programs to manipulate data and variables.
  - Arithmetic operators
  - Relational operators

- Logical operators
- Assignment operators
- Increment and decrement operators
- Conditional operators
- Bitwise operators
- Special operators

Java operators are classified into number of categories.

➢ ARITHMETIC OPERATORS

Arithmetic operators are used to construct mathematical expressions as in algebra

Operator	**Meaning**
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo division

RELATIONAL OPERATORS

❖ Compares two quantities depending on their relation.

❖ Java supports six relational operators.

Operator	**Meaning**
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
= =	is equal to
!=	is not equal to

A simple relational expression contains only one relational operator and is of the following form:

ae-1 relational operator ae-2

where **ae-1** and **ae-2** are arithmetic expressions

LOGICAL OPERATORS

Java has three logical operators.

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

A logical operator returns either TRUE or FALSE values.

Logical operator **&&** and **||** are used to check compound condition (**ie** for combining two or more relations)

When an expression combines two or more relational expressions then it is called logical expression or a compound relational expression

ASSIGNMENT OPERATORS

- Used to assign the value of an expression to a variable.
- Assignment operators are usually in the form "=".

v op=exp;

INCREMENT AND DECREMENT OPERATORS

❖ They are also called unary operator.

++ → Increment operator, add 1 to the operand

-- → Decrement operator, subtract 1 to the operand

They may also use to increment subscripted variables

CONDITIONAL OPERATORS

❖ The character pair **?:** is used for conditional operator.

❖ It is also called as **ternary** operator.

General Form **exp1 ? exp2: exp3**

where exp1, exp2, exp3 are expressions

The operator ?: works as follows

exp1 is evaluated first, if its is **true** then the **exp2** is evaluated.

If exp1 is **false**, **exp3** is evaluated

BITWISE OPERATORS:

❖ Bitwise operators are used to manipulate data at values of bit level.

These operators are used for testing the bits, or shifting them to the right or left.

❖ Bitwise operators may not to float or double.

Operator	Meaning
&	Bitwise AND
!	Bitwise OR
^	Bitwise exclusive OR
~	One's complement
<<	Shift left
>>	Shift right
>>>	Shift right with zero fill

SPECIAL OPERATORS

❖ Java supports special operators

➢ Instance of operator

➢ Dot operator (or) member selection operator (.)

✓ ***Instance of operator:***

❖ Instance of operator is an object reference operator.

❖ Allow us to determine whether the object belongs to a particular class or not.

❖ Return true, if the object on the left-hand side is an instance of the class given on the right-hand side.

Dot operator

The dot operator **(.)** is used to access the instance variables and methods of class objects.

It is also used to access classes and sub packages from a package.

CONTROL STATEMENTS

When a program breaks the sequential flow and jumps to another part of the code, it is called **branching.**

❖ When the branching is based on a particular condition, it is known as **conditional branching**.

❖ If branching takes place without any decision, it is known as **unconditional branching**. if statement

switch statement

Conditional operator statement

The following statements are known as control or decision making statements.

- if statement
- **switch** statement
- Conditional operator statement

IF Statement

The if statement is a powerful decision making statement and is used to **control the flow of execution of statements**.

General form **if (test expression)**

The expression is first evaluated.

❖ Depending on the value of the expression is true or false, control is transfer to a particular statement.

❖ The if statement are

1. simple **if** statement
2. **if…else** statement
3. Nested **if…else** statement
4. **else if** ladder

1. Simple If Statement

➢ If the test expression is **true** the **statement block will be executed**; otherwise the execution will **jump to the statement-x**

➢ Statement block may be single statement or a group of statement.

Example

General form

```
if (test expression)
{
statement-block;
}
statement-x;
```

Example

```
if (category == SPORTS)
{
        marks = marks +
bonus_marks;
}
System.out.println(marks);
```

2. **The If…Else Statement**

➢ If the test expression is **true**, then the **true-block statements are executed**.

➢ Otherwise, **the false block statements are executed**.

General form

```
if (test expression)
{
True block statements;
}
else
{
False block statements;
}
Statement-X;
```

Example

```
if (degree = = “BCA”)
{
        points = points+500;
}
else
{
        Points = points + 100;
}
```

3. Nesting of if …else statement

➢ Here if the **condition-1** is **false**, the **statement-3** will be executed; **otherwise** it **evaluates** the **condition-2**.

➢ If the **condition-2** is **true**, then **statement-1** will be executed; **otherwise** the **statement -2** will be **evaluated** and **then control** is **transferred** to the **statement-x**.

General form

```
if (test condition1)
{
        if (test condition2)
        {
                True
blockstatements-1;
        }
        else
        {
                False block
statement-2;
        }
}
else
{
        False block statements-3;
}
Statement-x;
```

Example

```
if (gender == “female”)
{
        if (balance>5000)
        {
                Bonus = 0.03 *
balance;
        }
        else
        {
                Bonus = 0.02 *
balance;
        }
}
else
{
        Bonus = 0.01 * balance;
}
balance=balance + bonus;
```

4. Else if ladder

Else If ladder is a chain of ifs in which the statement associated with each else is an if.

The condition is evaluated from the top to downwards.

➢ As soon as the **condition is true**, then **the statements associated with it are executed** and the **control is transferred to the statement -x.**

➢ When all the **n condition is false**, then the **final else containing** the **default-statement** will be executed.

General form

```
If (condition-1)
        statement-1;
else if (condition-2)
        statement -2;
else if (condition-3)
        statement -3;
…..
else if (condition n)
        statement -n;
else
        default-statement;
```

Example

```
If (marks>79)
        grade=”honors”;
else if (marks>79)
        grade=”first”;
else if (marks>79)
        grade=”second”;
else if (marks>79)
        grade=”third”;
else
        grade=”fail”; // Default-stmt
System.out.println(“grade=”+grade);
```

The Switch Statement

- It an **multiway** decision statement.
- The switch statement tests the value of a given variable against a list of **case** values.
- When a **match is found**, a block of statement associated with that **case** is executed.
- The expression is an integer expression or character known as **case labels**.
- Block1, block2 … are statements lists may contain zero or more statements.
- No need to put **braces** around **each block**
- Case labels end with a **colon (:)**
- The breakstatement at the end of each block signal the end of a particular case and causes an exit from the switch statement, transferring the control to the statement -x following the switch.
- The default is an option case; it will be executed if the value of the expression does not match with any of the case values.
- If not present, no action takes place when all matches fail and the control goes to the statement –x.

General form

```
switch(expression)
{
        case value-1:
                block-1
                break;
        case value-2:
                block-2
                break;
        case value-3:
                block-3
                break;
```

Example

```
switch(expression)
{
   case ‘1’:
      System.out.println(“Monday”);
      break;
   case ‘2’:
      System.out.println(“Tuesday”);
      break;
   case ‘3’:
      System.out.println(“Wednesday”);
      break;
```

```
        ……..
        …….
        default:
                default-block
                break;
    }
    statement-x;
```

```
    case '4':
        System.out.println("Thursday");
        break;
    ……
    ……..
    default:
    System.out.println ("WRONG
INPUT");
    break;
}
System.out.println(" WELCOME TO
THIS WEEK");
```

SIMPLE JAVA PROGRAM

Simple java program

```
class SampleOne
{
        public static void main (String args[])
        {
                System.out.println(" Java is better than C++");
        }
}
```

Class declaration

The first line

- Class SampleOne declares a class, java is a true object-oriented language and therefore, **everything must be placed inside a class**.
- **class** is a **keyword** and declares that a new class definition follows.
- SampleOne is a java identifier that specifies the name of the class to be defined.

Opening Brace

Every class definition in java begins with an opening brace **"{"** and ends with a matching closing brace **"}".**

The main line

The third line

public static void main (String args[])

- The above line defines a method named main.

- This is similar to the **main()** function in C/C++.
- Every java application program must include the **main() method**. This the starting point for the interpreter to begin the execution of the program.
- A java application can have any number of classes but **only one** of them must include a **main** method to initiate the execution.
- The line contains a number of keywords **public, static and void.**
- ***Public :*** The keyword public is an access specifier that declares the main method as unprotected and therefore making it accessible to all other classes.
- ***Static :*** *D*eclares this method as one that belongs to the entire class and not a part of any object of the class. The main methods must always be declared as static since the interpreter uses this method before any object are created.
- ***Void:*** The void states that the main method does not return any value.

The output line

The only executable statement in the program is

System.out.println(“Java is better than C++”);

This is similar to printf() statement of C or cout<< construct of C++.

Since java is a true object oriented language, every method must be part of an object.

The **println** method is a member of the **out** object, which is a static data member of **System** class.

This line prints the string “java is better than C++.”

CONSTRUCTORS

Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked.

- The main rule of constructors is that they should have the same name as the class.
- A class can have more than one constructor.
- They **does not return any value** and **do not specify even void**.
- Constructors are **automatically called** during the **creation of the objects**.

ADVANTAGES OF CONSTRUCTORS:

1. A constructor eliminates placing the default values.
2. A constructor eliminates calling the normal method implicitly.

TYPES OF CONSTRUCTORS:

- Based on creating objects in JAVA we have two types of constructors.
- They are

  Default/parameter less/no argument constructor and Parameterized constructor.

STRING AND STRING BUFFER CLASSES

Strings, which are widely used in Java programming, are a sequence of characters. In Java programming language, strings are treated as objects.

The Java platform provides the String class to create and manipulate strings.

String Class Methods

Method	**task performed**
s2=s1.toLowerCase;	Converts the string s1 to all lowercase
s2=s1.toUpperCase’;	Converts the string s1 to all uppercase
s2=s1.replace(‘x’,’y’);	Replace all appearances of x with y
s1.equal(s2);	Returns true if s1 is equal to s2
s2=s1.trim();	Remove white space at the beginning and end of the String s1
s1.equalsIgnoreCase(s2);	Returns true if s1 is equal to s2, ignoring the case of characters
s1.length();	Gives the length of s1.
s1.charAt(n)	Gives nth character of s1
s1.concat(s2);	Concatenates s1 and s2
s1.substring(n);	Gives substring starting from nth character.
s1.substirng(n,m);	Gives substring starting from nth character up to mth character
String.valueOf(p);	Creates a string object of the parameter p(simple type or object)

p.toString();	Creates a string representation of object p
s1.indexOf('x')	Gives the position of the first occurrence of 'x' in the string s1
s1.indexOf('x','n');	Gives the position 'x' that occurs after nth position in the string s1
String.valueOf(variable);	Converts the parameter value to string representation.
s1.compareTo(s2)	Returns negative if s1<s2, positive if s1>s2, zero if s1 and s2 equal.

StringBuffer class methods

StringBuffer class is a peer class of String.

- String creates strings of fixed_length
- StringBuffer class creates string of flexible length that can be modified in terms of both length and content.
- In stringbuffer class we can insert characters and substrings in the middle of a string, or append another string to the end.

1	public StringBuffer append(String s) Updates the value of the object that invoked the method. The method takes boolean, char, int, long, Strings, etc.
2	public StringBuffer reverse() The method reverses the value of the StringBuffer object that invoked the method.
3	public delete(int start, int end) Deletes the string starting from the start index until the end index.
4	public insert(int offset, int i) This method inserts a string s at the position mentioned by the offset.
5	replace(int start, int end, String str) This method replaces the characters in a substring of this StringBuffer with characters in the specified String.

SUMMARY

The provided document covers a comprehensive overview of Object-Oriented Concepts, Java History, Java Buzzwords, JVM Architecture, Data Types, Variables, Operators, Control Statements, Simple Java Program, Constructors, String and StringBuffer classes. Understand the basic Object-oriented concepts. Implement the basic constructs of Core Java.

ACTIVITIES

- Identify and list objects and their corresponding classes from everyday scenarios (e.g., car as an object, Car class).
- Create a hierarchy of classes related to vehicles (e.g., Vehicle -> Car, Truck) and demonstrate inheritance by sharing properties and methods.
- Implement method overloading and overriding using simple scenarios (e.g., Shape class with various subclasses like Circle, Square).
- Design a Java class that demonstrates encapsulation (private fields, public methods for access).
- Draw a diagram illustrating the components of the JVM and explain their functions.
- Simulate scenarios where students must use if-else statements or switch statements to control program flow based on different conditions.
- Create classes with multiple constructors and demonstrate how each constructor is invoked during object instantiation.

SELF-ASSESSMENT QUESTIONS

1. How would you define an object in your own words?
2. What are the key characteristics of an object? Can you provide examples?
3. How do state and behavior relate to objects? Can you illustrate this with a specific object?
4. Why do objects occupy space in memory, and how does this affect program performance?
5. How is a class like a blueprint? Can you provide a real-world analogy?
6. What types of data and methods are typically included in a class definition?
7. How does inheritance promote code reusability? Can you give an example?
8. What is polymorphism, and why is it a key concept in object-oriented programming?
9. What are the two main types of polymorphism in Java? Can you briefly explain each?
10. How does method overloading demonstrate polymorphism? Can you give a code example?
11. How does method overriding demonstrate polymorphism? Can you give a code example?

12. What is encapsulation, and how does it enhance data security in a program?
13. What are conditional statements, and why are they used in programming?
14. How do you structure an `if-else` statement to handle multiple conditions?
15. What is the purpose of the `else if` statement, and how does it differ from using multiple `if` statements?
16. What are the logical operators (`&&`, `||`, `!`) in Java, and how are they used in conditional statements?
17. How would you test the correctness of your conditional statements in your code?
18. What are the different types of constructors in Java? Explain each type briefly.
19. How does a parameterized constructor differ from a default constructor? Provide an example.
20. What is constructor overloading, and how is it implemented in Java?
21. What are the different types of loops available in Java? Describe each type.
22. What factors can affect the efficiency of a loop? How can you optimize looping statements?
23. How would you test and debug your loop structures to ensure they function correctly?

UNIT II

Objective: To equip the student with programming knowledge in Inheritance, Package, Interface and Exception handling

---

INHERITANCE

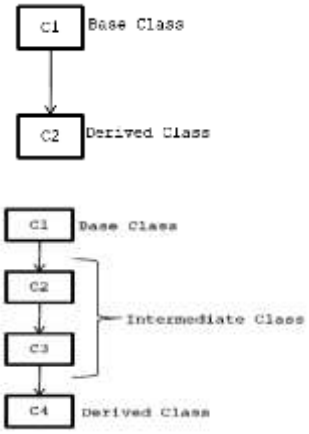
When one class acquires the properties of another class it is known as ***inheritance.***

- A class that is inherited is called a super class or base class.
- The class that does the inheriting is called a subclass or derived class.
- Advantage of inheritance is that it allows reusability of coding.

TYPES OF INHERITANCE

Inheritance may take different forms. They are

- Single inheritance (one super class, one sub class)
- Multilevel inheritance (derived from a derived class)
- Multiple Inheritance



Single inheritance

Single class is one in which there exists single base class and single derived class.

Multi level inheritance

Multi level inheritance is one which there exist single base class, single derived class and n number of intermediate base classes.

An intermediate base class is one, in one contest it acts as base class and in another context it acts as derived class.

Multiple Inheritance

Multiple inheritances are one supported by JAVA through classes but it is supported by JAVA through the concept of interfaces.

Defining a subclass

A subclass is defined as follows:

class subclassname extends superclassname

{
variables declaration;
methods declaration;
}

The keyword extends signifies that the properties of the super classname are extended to the subclassname.

SUBCLASS CONSTRUCTOR

- A subclass constructor is used to construct the instance variables of both the subclass and the super class.
- The sub class constructor uses the keyword super to invoke the constructor method of the super class.
- The keyword super is used in following conditions. Super may only be used within a subclass constructor method.
- The call to super class constructor must appear as the first statement within the sub class constructor.
- The parameter in the super call must match the order and type of the instance variable declared in the program.

MEMBER ACCESS RULES

The modifiers are also known as **access modifiers.**

Java provides three types of visibility modifiers: **public, private and protected.**

Public Access:

To declare the variable or method as public, it is visible to the entire class in which it is defined.

Example:

public int number;
public void sum ( ) {……..}

Friendly Access:

When no access modifier is specified, the number defaults to a limited version of public accessibility known as “friendly” level of access.

The difference between the “public” and “friendly” access is that the public modifier makes fields visible in all classes.
While friendly access makes fields visible only in the same package, but not in other package.

Protected Access:

The **protected** modifier makes the fields visible not only to all classes and subclasses in the same package but also to subclasses in other packages.
Non-subclasses in other packages cannot access the “protected” members.

Private Access:

Private fields are accessible only with their own class.
They cannot be inherited by subclasses and therefore not accessible in subclasses.
A method declared as **private** behaves like a method declared as **final.**

Private protected Access:

A field can be declared with two keywords **private** and **protected** together like:

private protected int codeNumber;
This gives a visibility level in between the “protected” access and “private” access.
Use **public** if the field is to be visible everywhere.
Use **protected** if the field is to be visible everywhere in the current package and also subclasses in other packages.
Use “default” if the field is to be visible everywhere in the current package only.

METHOD OVERLOADING

Method overloading is creating methods that have same name, but different parameter lists and different definitions. Method overloading is used when objects are required to perform similar tasks but using different input parameters.

When a method is called, java matches up the method name first and then the number and type of parameters to decide which one of the definitions to execute. This process is known as polymorphism.

Ex:

```
class room {
        float l,b;
        room(float x, float y)          {
                l=x;
                b=y;    }
        room(float x) {
                l = b = x; }
        int area() {
                return(l * b);   }
}
```

METHOD OVERRIDING

- A method defined in a super class is inherited by its sub class and is used by the objects created by the sub class.
- There may be occasions when we want an object to respond to the same method is called.
- This is possible by defining a method in the sub class that has the same name, same arguments and same return type as a method in the super class.
- When the methods are called, the method defined in the sub class is invoked and executed instead of the one in the super class. This is known as ***overriding.***

Example

```
class Super {
int x;
void display () {
System.out.println("Super x="+x);
System.out.println("Sub y="+y);
} }
class overridetest {
public static void main (String args[]) {
Sub S1=new Sub (100, 200);
S1.display (); }  }
```

```
Super (int x) {
this.x=x;
}
void display () {
System.out.println("x="+x);
} }

class Sub extends Super {
int y;
Sub (int x, int y) {
super (x);
this.y=y; } }
```

ABSTRACT METHODS AND CLASSES

In JAVA we have two types of classes. Concrete classes and abstract classes.

They are

- A concrete class is one which contains fully defined methods. Defined methods are also known as implemented or concrete methods. With respect to concrete class, we can create an object of that class directly.
- An abstract class is one which contains some defined methods and some undefined methods. Undefined methods are also known as unimplemented or abstract methods. Abstract method is one which does not contain any definition. To make the method as abstract we have to use a keyword called abstract before the function declaration**.**

ABSTRACT METHODS

- When a method is defined as final than that method is not re-defined in a subclass.
- Java allows a method to be re-defined in a sub class and those methods are called *abstract methods.*
- When a class contains one or more abstract methods, then it should be declared as abstract class.
- When a class is defined as abstract class, it must satisfy following conditions.
    - We can’t use abstract classes to instantiate objects directly. For example
      Op s = new Op( ) - is illegal because Op is an abstract class.
    - The abstract methods of an abstract class must be defined in its sub class.
    - We can’t declare abstract constructors or abstract static methods.
- Final allows the methods not redefine in the subclass.
- Abstract method must always be redefined in a subclass, thus making overriding compulsory.
- This is done using the modifier keyword abstract in the method definition.

FINAL VARIABLES AND METHODS

It prevents the subclasses form overriding the member of the superclass.

Final variables and methods are declared as final using the keyword **final** as a modifier.

Example: size =100

final int SIZE = 100;

final void showstatus( ){……..}

Making a method final ensures that the functionality defined in that method will never be altered in any way.

The value of a final variable can never be changed.

FINAL CLASSES

- A class that cannot be sub-classed is called a final class.
- It prevents a class being further sub-classed for security reasons.
- Any attempt to inherit these classes will cause an error.

```
final class Aclass
{
}
Final class Bclass extend someclass
{
}
```

FINALIZER METHODS

- Initialization→ constructor method is used to initialize an object when it is declared. This process is called initialization.
- Finalization→finalizer method is just opposite to initialization, it automatically frees up the memory resources used by the objects. This process is known as finalization.
- It acts like a destructor.
- The method can be added to any class.

The **finalize( )** method has this general form: 99

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

Here, the keyword **protected** is a specifier that prevents access to **finalize( )** by code defined outside its class.

PACKAGES

- A package is a collection of classes, interfaces and sub-packages.
- A sub-package in turns divides into classes, interfaces, sub-sub-packages, etc.
- Learning about JAVA is nothing but learning about various packages.
- By default one predefined package is imported for each and every JAVA program and whose name is **java.lang.*.**
- Whenever we develop any java program, it may contain many number of user defined classes and user defined interfaces.
- If we are not using any package name to place user defined classes and interfaces, JVM will assume its own package called NONAME package.
- In java we have two types of packages they are
  predefined or built-in or core packages and user or secondary or custom defined packages.

BENEFITS:

- The classes contained in the packages of other programs can be easily reused.
- In packages, classes can be unique compared with classes in other packages.
- That is two classes in two different packages can have the same name.
- They may be referred by their fully qualified name, comprising the package name and the class name.
- Packages provide a way to "hide" classes thus preventing other programs or packages from accessing classes that are meant for internal use only.
- Packages also provide a way for separating "design" form "coding".

USING PACKAGES

- Packages are organized in a hierarchical structure.
    - The package named java contains the package awt, which in turns contain various classes required for implementing graphical user interface. Best and easiest one to access the class
    - Used only once, not possible to access other classes of the package.

There are two ways of accessing the classes stored in a package

1. Using the fully qualified class name. (Using the package name containing the class and then appending the class name by using the dot operator.)

    E.g. java.awt.Color

2. Using the import statement, appear at the top of the file. Imported package class can be accessed anywhere in the program

Syntax:

```
import packagename.classname;
Or
import packagename.*
```

These are known as import statements and must appear at the top of the file, before any class declarations, **import** is a keyword.

- The first statement allows the specified class in the specified package to be imported. For example, the statement

```
import java.awt.Color;
```

imports the class **Color** and therefore the class name can now be directly used in the program.

- The second statement imports every class contained in the specified package. For example, the statement

```
import java.awt.*;
```

will bring all classes of java.awt package.

INTERFACES

Interfaces are basically used to develop user defined data types.

- With respect to interfaces we can achieve the concept of multiple inheritances.
- With interfaces we can achieve the concept of polymorphism, dynamic binding and hence we can improve the performance of a JAVA program in turns of memory space and execution time.
- An interface is a construct which contains the collection of purely undefined methods or an interface is a collection of purely abstract methods.

```
interface <InterfaceName>
{
variables declaration;
methods declaration;
}
```

Here, interface is the keyword.

Interface name represent a JAVA valid variable name.

Variables are declared as follows:

```
static final type VariableName = Value;
```

All variables are declared as constants. Methods declaration will contain only a list of methods without any body statements.

```
return-type methodName1 (parameter_list);
```

EXTENDING INTERFACES

- Interface can also be extended.
- An interface can be sub-interfaced from other interfaces.
- The new interface will inherit all the member of the super-interface.
- Interface can be extended using the keyword extends.

```
interface name2 extends name1
{
Body of name2
}
```

IMPLEMENTING INTERFACES

Implement the interface using implements keyword.

General form1	***General form2***
class classname **implements**	**class** classname **extends** supperclass
interfacename	**implements**
{	interface1,interface2,……..
body of classname	{
}	body of classname
	}

EXCEPTION HANDLING

Exceptions: **An exception is a condition that is caused by a run-time error in the program.**

- A java exception is an object that descirbes an error condition occurred in a piece of code. Exceptions can be generated by the Java run-time system or they can be manually generated by code.
- Exceptions thrown by a Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.
- Manually generated exceptions are typically used to report some error condition to the caller of a method.
- Java exception handling is managed via try, catch, throw, throw and finally.

Exception types

- All exceptions are subclasses of Throwable class. There are two subclasses that partition exception into two distinct branches, Exception and Error.
- Exception – this class is used for exceptional conditions that user program should catch. The class defined for custom exceptions are subclass to this class. A special subclass called RuntimeException, Exception of this class is automatically defined for a program.
- Error – This class defines exceptions that are not expected under normal circumstances. Exceptions of this type are used by the Java run-time system to indicate errors having to be deal by the run-time environment itself.

Syntax of Exception Handling Code

The basic concept of exception handling are throwing an exception and catching it. Program statements to be montiored are placed within try block. If an exception occurs within the try block, it is thrown. The code to handle the exception is placed within in the catch block. Any code that must be executed before a method returns is put in a finally block.

The general form of an exception-handling blcok:

```
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 before try block ends
}
```

- The try block can have one or more statements that could generate an exception. If any one statement generate an exception, the remaining statements in the block are skipped and execution jumps to the catch block.
- The catch block have one or more statements that are necessary to process the exception. Every try statement should be followed by atleast one catch statement.
- A try statement may have multiple catch blocks.
- A finally block is added to a try statement to handle an exception that is not handled or caught by the previous catch statements. A finally block can be

used to handle any exception within in a try block. A finally block may be added immediately after the try block or after the last catch block.

- When a finally block is defined, this is guaranteed to executed, regardless of whether or not an excpetion is thrown.

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

Throwing our own Exceptions

throw

An exception can be throw from the program. The general form is

throw ThrowableInstance;

where ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

The flow of execution stops immediately after the throw statement. The nearest enclosing try block is inspected whether it has a catch statement that matches the exception. If a match is found, control is transferred to that statement. If no matching catch is found, then the default exception handler halts the program and prints the stact trace.

Ex. throw new ArithmeticException();

throw new NumberFormatException();

throws

If a method can cause an exception but it doesn't handle then it must specify this behavior so that callers of the method can guard against that exception. This is indicated by using throws clause in the method declaration. The throws clasuse lists the type of exceptions that method might throw. The general form is

type method-name(parameter-list) throws exception-list

{

```
        // body of method
    }
```

where exception-list is a comma separated list of the exceptions that a method can throw.

Some common Java exceptions

Exception type	Cause of Exception
ArithmeticException	Caused by math errors
ArrayIndexOutOfBoundException	Caused by wrong array indexes
ArrayStoreException	Caused when a program tries to store the wrong type of data
FileNotFoundException	Caused by an attempt to access a nonexistent file.
IOException	Caused by general I/O failures
NullPointerException	Caused by referencing a null object
NumberFormatException	Caused when a conversion between strings and number fails
OutOfMemoryException	Caused when there's not enough memory to allocate a new object
SecurityException	Caused when an applet tries to perform an action not allowed by the browser's security setting
StringIndexOutOfBoundException	Caused when a program attempts to access a nonexistent character position in a string.

SUMMARY

Implement inheritance, packages, interfaces and exception handling of Core Java.

ACTIVITIES

- Create two classes, one as a superclass and the other as a subclass. Demonstrate inheritance by extending the superclass in the subclass.
- Discuss scenarios where each type of inheritance would be useful. Create example classes demonstrating each type of inheritance.

- Provide a superclass with a parameterized constructor. Create a subclass that inherits from it and initialize both superclass and subclass variables using the super keyword.
- Discuss scenarios where overriding is useful and what happens when a method is called using superclass and subclass references.
- Create an abstract class with one or more abstract methods. Provide concrete implementations in subclasses. Discuss why abstract classes are useful compared to concrete classes.
- Create a final class and demonstrate why it cannot be subclassed. Similarly, create a final method and discuss why it cannot be overridden.
- Create classes in different packages and demonstrate how access modifiers (public, protected, default, private) affect visibility across packages.
- Create interfaces with multiple methods. Implement these interfaces in classes and discuss how interfaces provide flexibility compared to classes.
- Provide classes with methods that throw exceptions. Demonstrate how exceptions propagate up the class hierarchy and how different exceptions can be handled using try-catch blocks.

SELF-ASSESSMENT QUESTIONS

1. What are the different types of inheritance supported in Java? Can you explain each type?
2. How do you define a superclass and a subclass in Java? What is the relationship between them?
3. What is method overriding, and how does it relate to inheritance? Provide an example.
4. How does inheritance facilitate polymorphism in Java?
5. How do the `public`, `protected`, and `private` access modifiers affect member access in inheritance?
6. What is a package in Java, and why is it used?
7. How do you implement an interface in a class? What is the syntax?
8. What are default and static methods in interfaces? How do they differ from regular interface methods?
9. How do you use try-catch blocks to handle exceptions? Provide a code example.
10. What is the purpose of the `finally` block in exception handling? When is it executed?
11. What is the difference between checked and unchecked exceptions? Can you give examples of each?

UNIT III

Objective: To enable the students to understand the concepts Multithreaded Programming and I/O handling.

MULTITHREADED PROGRAMMING

- **A flow of control is known as thread**.
- If a program contains **multiple flow of controls** for achieving concurrent execution then that program is **known as multithreaded program**.
- A program is said to be a multithreaded program if and only if in which there exist 'n' number of sub-programs there exist a separate flow of control.
- All such flow of controls are executing concurrently such flow of controls are known as threads and such type of applications or programs is called multithreaded programs.
- A thread is similar to a program that has a single flow of control. It has a beginning a body, and an end, and executes commands sequentially.

CREATING THREADS

Creating threads in Java is simple.

Threads are implemented in the form of objects that contain a method called run( ).

- The run( ) method is the heart and soul of any thread. It makes up the entire body of the thread and is the only method in which the thread's behavior can be implemented.
- A typical run( ) would appear as follows:

```
public void run( )
{
---- ( Statements for implementing thread )
----
}
```

The run( ) method should be invoked by an object of the concerned thread. This can be achieved by creating the thread and initiating it with the help of another thread method called start( ).

A NEW THREAD CAN BE CREATED IN TWO WAYS.

- **By creating a thread class :** Define a class that extends **thread** class and override its **run( )** method with the code required by the thread.
- **By converting a class to a thread**: Define a class that implements **Runnable** interface. The **Runnable** interface has only one method**, run( )** , that is to be defined in the method with the code to be executed by the thread.
- The approach to be used depends on what the class we are creating requires.
- If it requires to extend another class, then we have no choice but to implement the Runnable interface, since Java classes cannot have two superclasses.

EXTENDING THE THREAD CLASS

- We can make our class Runnable as a thread by extending the class **java.lang.Thread**. This gives us access to all the thread methods directly. It includes the following steps:
- Declare the class as extending the **Thread** class.
- Implement the **run( )** method that is responsible for executing the sequence of code that the thread will execute.
- Create a thread object and call the **start( )** method to initiate the thread execution.

Declaring the class

The thread class can be extended as follows:

```
class MyThread extends Thread
{
……..
……..
}
```

Now we have a new type of thread **MyThread.**

Implementing the run( ) method

- The run( ) method has been inherited by the class **MyThread.**
- We have to override this method in order to implement the code to be executed by our thread.

- The basic implementation of run( ) will look like this:

```
public void run( )
{
……..
…….. // Thread code here
}
```

when we start the new thread, Java calls the thread's run( ) method, so it is the run( ) where all the action takes place.

STARTING NEW THREAD

To actually create and run on instance of the thread class, we must write the following

MyThread aThread = new MyThread( );

aThread.start ( ); //invokes run( ) method

The second line calls the start( ) method causing the thread to move into the runnable state.

Then the Java runtime will schedule the thread to run by invoking its run( ) method. Now, the thread is said to be in the running state.

I/O STREAMS

A file is a collection of related records, a record is collection of fields and a filed is a group of characters. Storing and managing data using files is known as file processing which includes tasks such as creating files, updating files and manipulation of data. Java handles the file processing in the form of streams. Java also supports to write and read an object from the secondary storage device, known as object serialization.

Concepts of Streams

In file processing, input refers to the flow of data into a program and output means the flow of data out of a program.

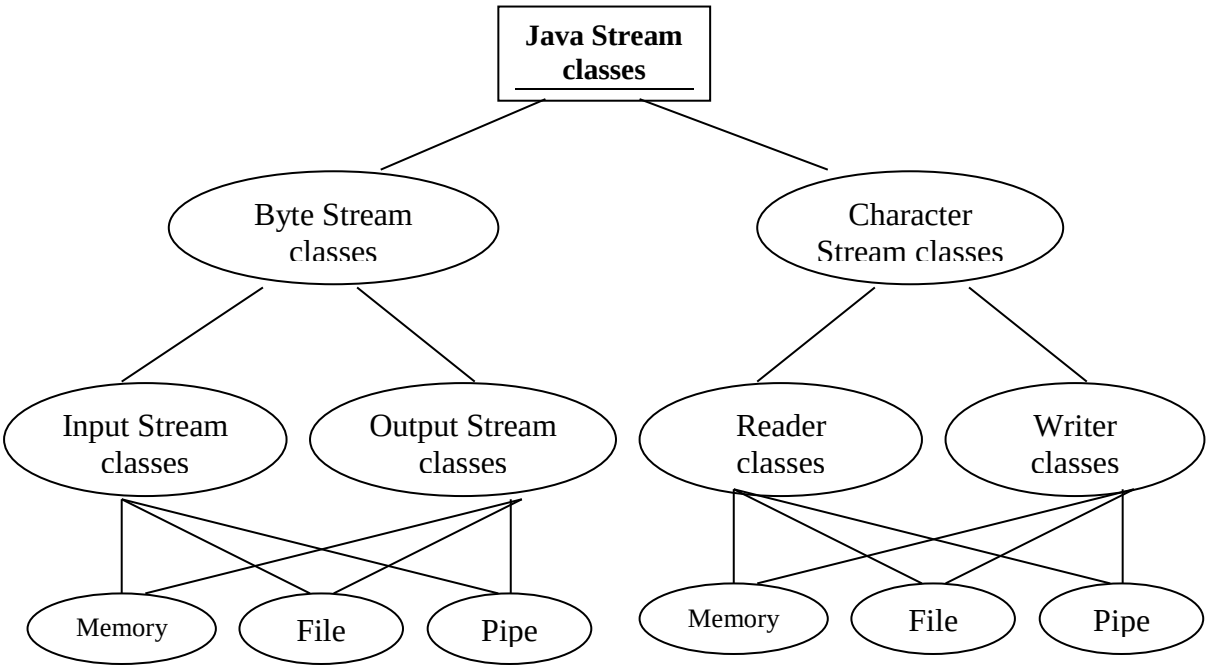
Java uses the concept of streams to represent the ordered sequence of data, a common characteristic shared by all the input/output devices.

A stream persent a uniform, easy-to-use, object-oriented interface between the program and the input/output devices.

A stream has a source and a destination, which may he physical devices or programs or other streams in the same program.

Java streams are classified into two basic types are input stream and output stream. An input stream extracts data from the source and sends it to the program. An output stream takes data from the program and sends it to the destination.

Stream Classes



The java.io package contains a large number of stream classes that provide capabilities for processing all types of data. These classes are classified into two categories based on the type of data they can process.

1. Byte stream classes that provide support for handling I/O operations on bytes.
2. Character stream classes that provide support for managing I/O operations on characters.

Byte stream and character stream classes contain specialized classes to deal with input and output operations independently on various types of devices.

Byte Stream Classes

Byte stream classes provide functional features for creating and manipulating streams and files for reading and writing bytes. Java provides two kinds of byte stream classes input stream and output stream classes.

Input stream classes: Input stream classes are used to read bytes, include a super class known as InputStream and various subclasses for supporting various input-related functions.

The super class InputStream is an abstract class. The InputStream class defines methods for performing input functions such as

- Reading bytes
- Closing streams
- Marking positions in streams
- Skipping ahead in a stream
- Finding the number of bytes in a stream

Some of the InputStream methods are

1. read( )	Reads a byte from the input stream
2. read(byte b[ ])	Reads an array of bytes into b
3. read(byte b[ ], int n, int m)	Reads m bytes into b starting from nth byte.
4. available( )	Gives number of bytes available in the input
5. skip(n)	Skips over n bytes from the input stream
6. reset( )	Goes back to the beginning of the stream
7. close( )	Close the input stream

The DataInputStream class extends FilterInputStream and implements the interface DataInput. The DataInput interface contains the following methods:

readShort( )	readDouble( )	readInt( )
readLine( )	readLong( )	readChar( )
readFloat( )	readBoolean( )	readUTF( )

Output Stream Classes: Output stream classes are derived from the base clas OutputStream. The OutputStream is an abstract class. The subclasses of the OutputStream can be used for performing the output operations.

The OutputStream includes methods to perform the following operations

1. Writing bytes
2. Closing streams
3. Flusing streams

Some of the methods defined in the OutputStream class.

Method	Description
1. write( )	Writes a byte to the output stream
2. write(byte[ ] b)	Writes all bytes in the array b to the output stream
3. write(byte b[ ], int n, int m)	Writes m bytes from array b starting from nth byte
4. close( )	Closes the output stream
5. flush( )	Flushes the output stream

The DataOutputStream implements the interface DataOutput. The DataOutput interface consist the following methods.

writeShort( ) writeDouble( ) writeInt( )
writeLine( ) writeLong( ) writeChar( )
writeFloat( ) writeBoolean( ) writeUTF( )

Character Stream classes

Character streams can be used to read and write 16-bit unicode characters. There are two kinds of character stream classes,

1. reader stream classes and
2. writer stream classes.

Reader Stream classes: Reader stream classes are designed to read character from the files. The Reader class is an abstract class and it is the base class for all other classes in this group. The Reader class contains methods to handle various input operations based on characters.

Writer Stream classes: The writer stream classes are designed to perform all output operations on files. These classes do the task based on characters. The Writer class is an abstract class and it is the base for all the classes in this group.

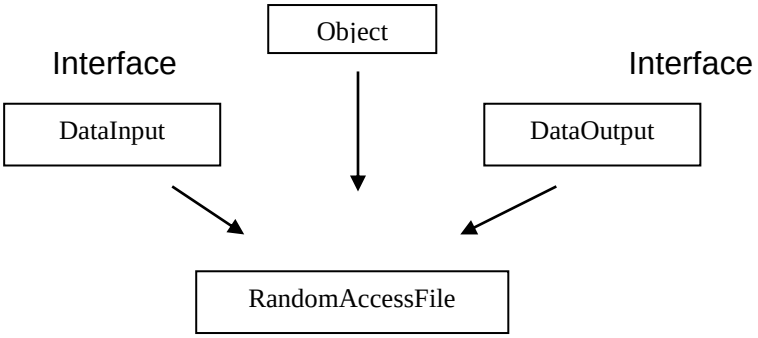
The classes used for various input/output operations in both groups are given below.

Task	Character Stream Class	Byte Stream class
Performing input operations	Reader	InputStream
Buffering input	BufferedReader	BufferedInputStream
Keeping track of line numbers	LineNumberReader	LineNumberInputStream
Reading from an array	CharArrayReader	ByteArrayInputStream
Reading from files	FileReader	FileInputStream
Filtering the input	FilterReader	FilterInputStream
Puhsing back	PushbackReader	PushbackInputStream
Reading from a pipe	PipedReader	PipedInputStream
Reading from a string	StringReader	StringBufferedInputStream
Reading primitive types		DataInputStream
Performing output operations	Writer	OutputStream
Buffering output	BufferedWriter	BufferedOutputStream
Writing to an array	CharArrayWriter	ByteArrayOutputStream
Filtering the output	FilterWriter	FilterOutputStream
Writing to a file	FileWriter	FileOutputStream
Writing to a pipe	PipedWriter	PipedOutputStream
Writing to a string	StringWriter	
Writing primitive types		DataOutputStream

Other usefule I/O classes

The java.io pacakge supports many other classes for performing certain specialized functions. Some of the classes are

- RandomAccessFile – this class enables to read and write bytes, text and Java data types to any location in the file. This class extends object class and implements DataInput and DataOutput interfaces. This class implements methods of both interfaces.



Implementation of the RandomAccessFile

- StreamTokenizer – this class is a subclass of object can be used for breaking up a stream of text from an input text file into meaningful pieces called tokens.

Other Stream classes

- Object Streams – Java supports input and output operations on objects using the object streams. The object streams are created using the ObjectInputStream and ObjectOutputStream classes. The contents of the file is read or write as object, and this process is called object serialization.
- Piped Streams – Piped streams provide functionality for threads to communicate and exchange data between them.
- Pushback Streams – the pushback streams are created by the classes PushbackInputStream or PushbackReader, can be used to push a single byte or character back into the input stream so that it can be reread. This is commonly used with parsers.
- Filtered Streams – These streams provide the basic capability to create input and output streams for filtering input/output in a number of ways. These streams placed between an input stream and an output stream and perform some optional processing on the data they transfer. Java supports this through two abstract classes, FilterInputStream and FilterOutputStream.

SUMMARY

Implement multi-threading and I/O Streams of Core Java

ACTIVITIES

- Implement two classes ThreadClassA and ThreadClassB to demonstrate thread creation using both extending the Thread class and implementing the Runnable interface.
- Create a shared resource accessed by multiple threads to demonstrate synchronization.
- Implement a producer-consumer scenario using wait() and notify() methods for inter-thread communication.

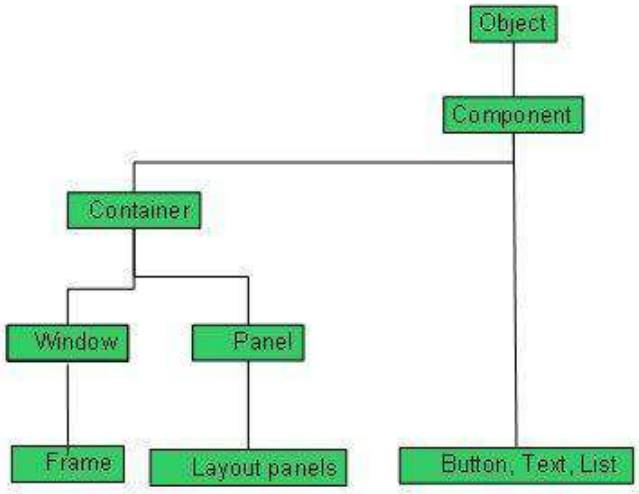
SELF ASSESSMENT QUESTIONS

1. What is a thread in Java, and how does it differ from a process?
2. How do you create a thread in Java? Explain both the `Runnable` interface and the `Thread` class.
3. Can you describe the thread lifecycle and the methods that change a thread's state?
4. What is synchronization, and why is it necessary in multithreaded programming?
5. Explain the concept of a synchronized block and how it differs from a synchronized method.
6. What is the purpose of the `wait()`, `notify()`, and `notifyAll()` methods?
7. How do you implement producer-consumer problems using these methods?
8. What is the difference between thread-safe and non-thread-safe classes?
9. How can you make a class thread-safe in Java?
10. What are the main types of I/O in Java (byte stream vs. character stream)?
11. How do you read from and write to files in Java using I/O streams?
12. What is the purpose of `BufferedReader` and `BufferedWriter`?
13. What is serialization in Java, and why is it used?
14. How do you implement serialization for a custom class?
15. How does Java handle exceptions in I/O operations?
16. What are checked and unchecked exceptions in the context of I/O?

- **UNIT IV**

Objective: To enable the students to use AWT controls, Event Handling and Swing for GUI.

AWT CONTROLS



Every AWT controls inherits properties from Component class.

COMPONENT

The class **Component** is the abstract base class for the non menu user-interface controls of AWT. Component represents an object with graphical representation.

CONTAINER

The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container such as **Frame, Dialog** and **Panel**.

It is basically a screen where the where the components are placed at their specific locations. Thus it contains and controls the layout of components.

Types of containers:

There are four types of containers in Java AWT:

1. Window
2. Panel
3. Frame
4. Dialog

Window

The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window. We need to create an instance of Window class to create this container.

Panel

The Panel is the container that doesn't contain title bar, border or menu bar. It is generic container for holding the components. It can have other components like button, text field etc. An instance of Panel class creates a container, in which we can add components.

Frame

The Frame is the container that contain title bar and border and can have menu bars. It can have other components like button, text field, scrollbar etc. Frame is most widely used container while developing an AWT application.

LABEL

Label is a passive control because it does not create any event when accessed by the user. The label control is an object of Label. A label displays a single line of read-only text. However the text can be changed by the application programmer but cannot be changed by the end user in any way.

Field

- static int CENTER -- Indicates that the label should be centered.
- static int LEFT -- Indicates that the label should be left justified.
- static int RIGHT -- Indicates that the label should be right justified.

Constructors

Label() - Constructs an empty label.
Label(String text) - Constructs a new label with the specified string of text, left justified.
Label(String text, int alignment) - Constructs a new label that presents the specified string of text with the specified alignment.

Methods

void setText(String text)	It sets the texts for label with the specified text.
void setAlignment(int alignment)	It sets the alignment for label with the specified alignment.
String getText()	It gets the text of the label
int getAlignment()	It gets the current alignment of the label.
void addNotify()	It creates the peer for the label.
protected String paramString()	It returns the string the state of the label.

BUTTON

A button is basically a control component with a label that generates an event when pushed. The **Button** class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed.

When we press a button and release it, AWT sends an instance of **ActionEvent** to that button by calling **processEvent** on the button. The **processEvent** method of the button receives the all the events, then it passes an action event by calling its own method **processActionEvent**. This method passes the action event on to action listeners that are interested in the action events generated by the button.

To perform an action on a button being pressed and released, the **ActionListener** interface needs to be implemented. The registered new listener can receive events from the button by calling **addActionListener** method of the button. The Java application can use the button's action command as a messaging protocol.

CONSTRUCTOR

Button( )	It constructs a new button with an empty string i.e. it has no label.
Button (String text)	It constructs a new button with given string as its label.

Methods

void setText (String text)	It sets the string message on the button
String getText()	It fetches the String message on the button.
void setLabel (String label)	It sets the label of button with the specified string.
String getLabel()	It fetches the label of the button.
void addActionListener(ActionListener l)	It adds the specified action listener to get the action events from the button.
String getActionCommand()	It returns the command name of the action event fired by the button.
ActionListener[ ] getActionListeners()	It returns an array of all the action listeners registered on the button.
protected String paramString()	It returns the string which represents the state of button.
void removeActionListener (ActionListener l)	It removes the specified action listener so that it no longer receives action events from the button.
void setActionCommand(String command)	It sets the command name for the action event given by the button.

TEXT FIELD

The object of a **TextField** class is a text component that allows a user to enter a single line text and edit it. It inherits **TextComponent** class, which further inherits **Component** class.

When we enter a key in the text field (like key pressed, key released or key typed), the event is sent to **TextField**. Then the **KeyEvent** is passed to the registered **KeyListener**. It can also be done using **ActionEvent**; if the ActionEvent is enabled on the text field, then the ActionEvent may be fired by pressing return key. The event is handled by the **ActionListener** interface.

Constructor

TextField()	It constructs a new text field component.
TextField(String text)	It constructs a new text field initialized with the given string text to be displayed.
TextField(int columns)	It constructs a new textfield (empty) with given number of columns.
TextField (String text, int columns)	It constructs a new text field with the given text and given number of columns (width).

Methods

void addNotify()	It creates the peer of text field.
boolean echoCharIsSet()	It tells whether text field has character set for echoing or not.
void addActionListener(ActionListener l)	It adds the specified action listener to receive action events from the text field.
ActionListener[] getActionListeners()	It returns array of all action listeners registered on text field.

AccessibleContext getAccessibleContext()	It fetches the accessible context related to the text field.
int getColumns()	It fetches the number of columns in text field.
char getEchoChar()	It fetches the character that is used for echoing.
void removeActionListener (ActionListener l)	It removes specified action listener so that it doesn't receive action events anymore.
void setColumns(int columns)	It sets the number of columns in text field.
void setEchoChar(char c)	It sets the echo character for text field.
void setText(String t)	It sets the text presented by this text component to the specified text.

TEXT AREA

The object of a TextArea class is a multiline region that displays text. It allows the editing of multiple line text. It inherits TextComponent class.

The text area allows us to type as much text as we want. When the text in the text area becomes larger than the viewable area, the scroll bar appears automatically which helps us to scroll the text up and down, or right and left.

Class constructors:

TextArea()	It constructs a new and empty text area with no text in it.
TextArea (int row, int column)	It constructs a new text area with specified number of rows and columns and empty string as text.
TextArea (String text)	It constructs a new text area and displays the specified text in it.
TextArea (String text, int row, int column)	It constructs a new text area with the specified text in the text area and specified number of rows and

	columns.
TextArea (String text, int row, int column, int scrollbars)	It construcst a new text area with specified text in text area and specified number of rows and columns and visibility.

Methods

void addNotify()	It creates the peer of text field.
boolean echoCharIsSet()	It tells whether text field has character set for echoing or not.
void addActionListener(ActionListener l)	It adds the specified action listener to receive action events from the text field.
ActionListener[] getActionListeners()	It returns array of all action listeners registered on text field.
AccessibleContext getAccessibleContext()	It fetches the accessible context related to the text field.
int getColumns()	It fetches the number of columns in text field.
char getEchoChar()	It fetches the character that is used for echoing.
void removeActionListener (ActionListener l)	It removes specified action listener so that it doesn't receive action events anymore.
void setColumns(int columns)	It sets the number of columns in text field.
void setEchoChar(char c)	It sets the echo character for text field.
void setText(String t)	It sets the text presented by this text component to the specified text.

CHECKBOX

The Checkbox class is used to create a checkbox. It is used to turn an option on (true) or off (false). Clicking on a Checkbox changes its state from "on" to "off" or from "off" to "on".

Constructor

Checkbox()	It constructs a checkbox with no string as the label.
Checkbox(String label)	It constructs a checkbox with the given label.
Checkbox(String label, boolean state)	It constructs a checkbox with the given label and sets the given state.
Checkbox(String label, boolean state, CheckboxGroup group)	It constructs a checkbox with the given label, set the given state in the specified checkbox group.
Checkbox(String label, CheckboxGroup group, boolean state)	It constructs a checkbox with the given label, in the given checkbox group and set to the specified state.

Method

void addItemListener(ItemListener IL)	It adds the given item listener to get the item events from the checkbox.
AccessibleContext getAccessibleContext()	It fetches the accessible context of checkbox.
void addNotify()	It creates the peer of checkbox.
CheckboxGroup getCheckboxGroup()	It determines the group of checkbox.
ItemListener[] getItemListeners()	It returns an array of the item listeners registered on checkbox.

String getLabel()	It fetched the label of checkbox.
Object[] getSelectedObjects()	It returns an array (size 1) containing checkbox label and returns null if checkbox is not selected.
boolean getState()	It returns true if the checkbox is on, else returns off.
protected void processItemEvent (ItemEvent e)	It process the item events occurring in the checkbox by dispatching them to registered ItemListener object.
void removeItemListener(ItemListener l)	It removes the specified item listener so that the item listener doesn't receive item events from the checkbox anymore.
void setCheckboxGroup(CheckboxGroup g)	It sets the checkbox's group to the given checkbox.
void setLabel(String label)	It sets the checkbox's label to the string argument.
void setState(boolean state)	It sets the state of checkbox to the specified state.

CHECKBOXGROUP

The object of CheckboxGroup class is used to group together a set of Checkbox. At a time only one check box button is allowed to be in "on" state and remaining check box button in "off" state. It inherits the object class.

CHOICE

The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits Component class.

Constructor

Choice()	It constructs a new choice menu.

Methods

void add(String item)	It adds an item to the choice menu.
void addItemListener(ItemListener l)	It adds the item listener that receives item events from the choice menu.
void addNotify()	It creates the peer of choice.
String getItem(int index)	It gets the item (string) at the given index position in the choice menu.
int getItemCount()	It returns the number of items of the choice menu.
ItemListener[] getItemListeners()	It returns an array of all item listeners registered on choice.
T[] getListeners(Class listenerType)	Returns an array of all the objects currently registered as FooListeners upon this Choice.
int getSelectedIndex()	Returns the index of the currently selected item.
String getSelectedItem()	Gets a representation of the current choice as a string.
Object[] getSelectedObjects()	Returns an array (length 1) containing the currently selected item.
void insert(String item, int index)	Inserts the item into this choice at the specified position.
void remove(int position)	It removes an item from the choice menu at the given index position.

void remove(String item)	It removes the first occurrence of the item from choice menu.
void removeAll()	It removes all the items from the choice menu.
void removeItemListener (ItemListener l)	It removes the mentioned item listener. Thus is doesn't receive item events from the choice menu anymore.
void select(int pos)	It changes / sets the selected item in the choice menu to the item at given index position.
void select(String str)	It changes / sets the selected item in the choice menu to the item whose string value is equal to string specified in the argument.

LIST

The object of List class represents a list of text items. With the help of the List class, user can choose either one item or multiple items. It inherits the Component class.

Constructor

List()	It constructs a new scrolling list.
List(int row_num)	It constructs a new scrolling list initialized with the given number of rows visible.
List (int row_num, Boolean multipleMode)	It constructs a new scrolling list initialized which displays the given number of rows.

Methods

void add(String item)	It adds the specified item into the end of scrolling list.
void add(String item, int index)	It adds the specified item into list at the

	given index position.
void addActionListener(ActionListener l)	It adds the specified action listener to receive action events from list.
void addItemListener(ItemListener l)	It adds specified item listener to receive item events from list.
void addNotify()	It creates peer of list.
void deselect(int index)	It deselects the item at given index position.
ActionListener[] getActionListeners()	It returns an array of action listeners registered on the list.
String getItem(int index)	It fetches the item related to given index position.
int getItemCount()	It gets the count/number of items in the list.
ItemListener[] getItemListeners()	It returns an array of item listeners registered on the list.
String[] getItems()	It fetched the items from the list.
int getRows()	It fetches the count of visible rows in the list.
int getSelectedIndex()	It fetches the index of selected item of list.
int[] getSelectedIndexes()	It gets the selected indices of the list.
String getSelectedItem()	It gets the selected item on the list.
String[] getSelectedItems()	It gets the selected items on the list.
Object[] getSelectedObjects()	It gets the selected items on scrolling list in

	array of objects.
int getVisibleIndex()	It gets the index of an item which was made visible by method makeVisible()
void makeVisible(int index)	It makes the item at given index visible.
boolean isIndexSelected(int index)	It returns true if given item in the list is selected.
boolean isMultipleMode()	It returns the true if list allows multiple selections.
protected String paramString()	It returns parameter string representing state of the scrolling list.
protected void processEvent(AWTEvent e)	It process the events on scrolling list.
protected void processItemEvent(ItemEvent e)	It process the item events occurring on list by dispatching them to a registered ItemListener object.
void removeActionListener (ActionListener l)	It removes specified action listener. Thus it doesn't receive further action events from the list.
void removeItemListener(ItemListener l)	It removes specified item listener. Thus it doesn't receive further action events from the list.
void remove(int position)	It removes the item at given index position from the list.
void remove(String item)	It removes the first occurrence of an item from list.
void removeAll()	It removes all the items from the list.

void replaceItem(String newVal, int index)	It replaces the item at the given index in list with the new string specified.
void select(int index)	It selects the item at given index in the list.
void setMultipleMode(boolean b)	It sets the flag which determines whether the list will allow multiple selection or not.
void removeNotify()	It removes the peer of list.

PANEL

The Panel is a simplest container class. It provides space in which an application can attach any other component. It inherits the Container class.

It doesn't have title bar.

MENU

The object of MenuItem class adds a simple labeled menu item on menu. The items used in a menu must belong to the MenuItem or any of its subclass.

The object of Menu class is a pull down menu component which is displayed on the menu bar. It inherits the MenuItem class.

WORKING WITH FRAME CLASS

The class **Frame** is a top level window with border and title. It uses BorderLayout as default layout manager.

Constructor

Frame() Constructs a new instance of Frame that is initially invisible.
Frame(GraphicsConfiguration gc) Constructs a new, initially invisible Frame with the specified GraphicsConfiguration.
Frame(String title) Constructs a new, initially invisible Frame object with the specified title.

Frame(String title, GraphicsConfiguration gc) Constructs a new, initially invisible Frame object with the specified title and a GraphicsConfiguration.

Methods

void addNotify() Makes this Frame displayable by connecting it to a native screen resource.
int getCursorType() Deprecated. As of JDK version 1.1, replaced by Component.getCursor().
int getExtendedState() Gets the state of this frame.
static Frame[] getFrames() Returns an array of all Frames created by this application.
Image getIconImage() Returns the image to be displayed as the icon for this frame.
Rectangle getMaximizedBounds() Gets maximized bounds for this frame.
MenuBar getMenuBar() Gets the menu bar for this frame.
int getState() Gets the state of this frame (obsolete).
String getTitle() Gets the title of the frame.
boolean isResizable() Indicates whether this frame is resizable by the user.

boolean isUndecorated() Indicates whether this frame is undecorated.
protected String paramString() Returns a string representing the state of this Frame.
void remove(MenuComponent m) Removes the specified menu bar from this frame.
void removeNotify() Makes this Frame undisplayable by removing its connection to its native screen resource.
void setCursor(int cursorType) Deprecated. As of JDK version 1.1, replaced by Component.setCursor(Cursor).
void setExtendedState(int state) Sets the state of this frame.
void setIconImage(Image image) Sets the image to be displayed as the icon for this window.
void setMaximizedBounds(Rectangle bounds) Sets the maximized bounds for this frame.
void setMenuBar(MenuBar mb) Sets the menu bar for this frame to the specified menu bar.
void setResizable(boolean resizable) Sets whether this frame is resizable by the user.
void setState(int state) Sets the state of this frame (obsolete).
void setTitle(String title) Sets the title for this frame to the specified string.

FONTS

In Java, **Font** is a class that belongs to the **java.awt** package. It implements the Serializable interface. FontUIResource is the direct known subclass of the Java Font class.

It represents the font that are used to render the text. In Java, there are two technical terms that are used to represent font are **characters** and **Glyphs**.

There are two types of fonts in Java:

- Physical Fonts
- Logical Fonts

Physical Fonts

Physical fonts are actual Java font library. It contains tables that maps character sequence to glyph sequences by using the font technology such as **TrueType Fonts** (TTF) and **PostScript Type 1 Font**. Note that all implementation of Java must support TTF. Using other font technologies is implementation dependent. Physical font includes the name such as **Helvetica, Palatino, HonMincho**, other font names.

The property of the physical font is that it uses the limited set of writing systems such as **Latin characters** or only **Japanese** and **Basic Latin** characters. It may vary as to configuration changes. If any application requires a specific font, user can bundle and instantiate that font by using the **createFont()** method of the Java Font class.

Logical Fonts

Java defines **five** logical font families that are **Serif, SansSerif, Monospaced, Dialog,** and **DialogInput**. It must be supported by the JRE. Note that JRE maps the logical font names to physical font because these are not the actual font libraries. Usually, mapping implementation is locale dependent. Each logical font name map to several physical fonts in order to cover a large range of characters.

A Font object have three different names that are:

- **Logical font name:** It is the name that is used to construct the font.
- **Font face name:** It is the name of particular font face. For example, Helvetica Bold.

- **Family name:** It is the name of the font family. It determines the typograph design among several faces.

LAYOUT MANAGER

The Layout managers enable us to control the way in which visual components are arranged in the GUI forms by determining the size and position of components within the containers.

Types of LayoutManager

There are 4 layout managers in Java

- **FlowLayout**: It arranges the components in a container like the words on a page. It fills the top line from **left to right and top to bottom**. The components are arranged in the order as they are added i.e. first components appears at top left, if the container is not wide enough to display all the components, it is wrapped around the line. Vertical and horizontal gap between components can be controlled. The components can be **left, center or right aligned.**
- **BorderLayout**: It arranges all the components along the edges or the middle of the container i.e. **top, bottom, right and left** edges of the area. The components added to the top or bottom gets its preferred height, but its width will be the width of the container and also the components added to the left or right gets its preferred width, but its height will be the remaining height of the container. The components added to the center gets neither its preferred height or width. It covers the remaining area of the container.
- **GridLayout**: It arranges all the components in a grid of **equally sized cells**, adding them from the **left to righ**t and **top to bottom**. Only one component can be placed in a cell and each region of the grid will have the same size. When the container is resized, all cells are automatically resized. The order of placing the components in a cell is determined as they were added.
- **GridBagLayout**: It is a powerful layout which arranges all the components in a grid of cells and maintains the aspect ration of the object whenever the container is resized. In this layout, cells may be different in size. It assigns a consistent horizontal and vertical gap among components. It allows us to specify a default alignment for components within the columns or rows.

EVENT HANDLING

Event and Listener (Java Event Handling)

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc. The java.awt.event package provides many event classes and Listener interfaces for event handling.

Java Event classes and Listener interfaces

Event Classes	Listeners
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Following steps are required to perform event handling:

1. Register the component with the Listener
2. Java Event Handling Code

We can put the event handling code into one of the following places:

- Within class
- Other class
- Anonymous class

EVENT DELEGATION MODEL

The **Delegation Event Model in Java** is based on a hierarchy of objects, with one object acting as the source of the event and other objects acting as listeners that respond to those events. When an event occurs, the source object generates an event object and passes it to all the registered listeners. Each listener then has the opportunity to process the event and respond accordingly.

One of the key benefits of the Delegation Event Model in Java is that it allows for a high level of modularity in Java applications. By separating the event source from the event listener, you can create more flexible and scalable code. This model also allows for easy customization, as different event listeners can be created to handle different types of events.

The Delegation Event Model in Java consists of three main components: event source, event object, and event listener.

Event source: This is the object that generates the event. When an event occurs, the event source creates an event object and passes it to all registered event listeners. Examples of event sources can be a button, a text field, or any other component of the user interface.

Event object: This is a Java object that encapsulates information about the event that occurred. It contains details such as the type of event, the source of the event, and any additional data that may be relevant to the event. The event object is passed to all registered event listeners, allowing them to respond to the event appropriately.

Event listener: This is an interface that defines methods for responding to events. To handle an event, an object must implement the appropriate event listener interface and register itself with the event source. When the event occurs, the event

source calls the appropriate method on each registered event listener, passing in the event object.

HANDLING MOUSE AND KEYBOARD EVENTS

Mouse Events

MouseListener and MouseMotionListener is an interface in java.awt.event package . Mouse events are of two types. MouseListener handles the events when the mouse is not in motion. While MouseMotionListener handles the events when mouse is in motion.

There are five types of events that MouseListener can generate.

There are five abstract functions that represent these five events.

The abstract functions are:

1. void mouseReleased(MouseEvent e) : Mouse key is released
2. void mouseClicked(MouseEvent e) : Mouse key is pressed/released
3. void mouseExited(MouseEvent e) : Mouse exited the component
4. void mouseEntered(MouseEvent e) : Mouse entered the component
5. void mousepressed(MouseEvent e) : Mouse key is pressed

There are two types of events that MouseMotionListener can generate. There are two abstract functions that represent these five events. The abstract functions are:

1. **void mouseDragged(MouseEvent e)** : Invoked when a mouse button is pressed in the component and dragged. Events are passed until the user releases the mouse button.
2. **void mouseMoved(MouseEvent e)** : invoked when the mouse cursor is moved from one point to another within the component, without pressing any mouse buttons.

Keyboard Events

The Java KeyListener is notified whenever you change the state of key. It is notified against KeyEvent. The KeyListener interface is found in java.awt.event package, and it has three methods.

Interface declaration

Following is the declaration for **java.awt.event.KeyListener** interface:

1. **public interface** KeyListener **extends** EventListener

Methods of KeyListener interface

The signature of 3 methods found in KeyListener interface are given below:

public abstract void keyPressed (KeyEvent e);	It is invoked when a key has been pressed.
public abstract void keyReleased (KeyEvent e);	It is invoked when a key has been released.
public abstract void keyTyped (KeyEvent e);	It is invoked when a key has been typed.

ADAPTER CLASSES

In Java's event handling mechanism, adapter classes are abstract classes provided by the Java AWT (Abstract Window Toolkit) package for receiving various events. These classes contain empty implementations of the methods in an event listener interface, providing a convenience for creating listener objects.

The adapter classes in Java are

- WindowAdapter
- KeyAdapter
- MouseAdapter
- FocusAdapter
- ContainerAdapter
- ComponentAdapter

These adapter classes implement interfaces like WindowListener, KeyListener, MouseListener, FocusListener, ContainerListener, and ComponentListener respectively, which contain methods related to specific events.

INNER CLASSES

A Java inner class is a class that is defined inside another class. The concept of inner class works with nested Java classes where outer and inner classes are used.

The main class in which inner classes are defined is known as the outer class and all other classes which are inside the outer class are known as Java inner classes.

SUMMARY

Implement AWT and Event handling.

ACTIVITIES

- Represents an object with graphical representation.

SELF–ASSESSMENT QUESTIONS

1. What is AWT, and how does it differ from Swing?
2. What are the primary components provided by AWT?
3. What is the purpose of the `Frame` class in AWT?
4. What are layout managers in AWT? Name and explain a few common ones.
5. How do you set a layout manager for a container in AWT?
6. How do you perform custom painting in AWT?
7. What is the role of the `Graphics` class in AWT?
8. What is Swing, and how does it enhance AWT?
9. Explain the concept of "lightweight" components in Swing.
10. What is the "look and feel" of a Swing application, and how can it be changed?
11. How do you set a custom look and feel for a Swing application?
12. Discuss the advantages and disadvantages of using Swing over AWT.
13. Can you use AWT components in a Swing application? If so, how?
14. What is event handling in Java?
15. What is an event listener, and how do you implement one in Java?
16. How do you use ActionListener, MouseListener, and KeyListener in your applications?
17. How do you register an event listener with a component?
18. What is the Event Dispatch Thread, and why is it important in Swing?
19. How do you create and handle custom events in Java?
20. What is the difference between using built-in events and custom events?

UNIT V

Objective: To equip the student with programming knowledge SWING components.

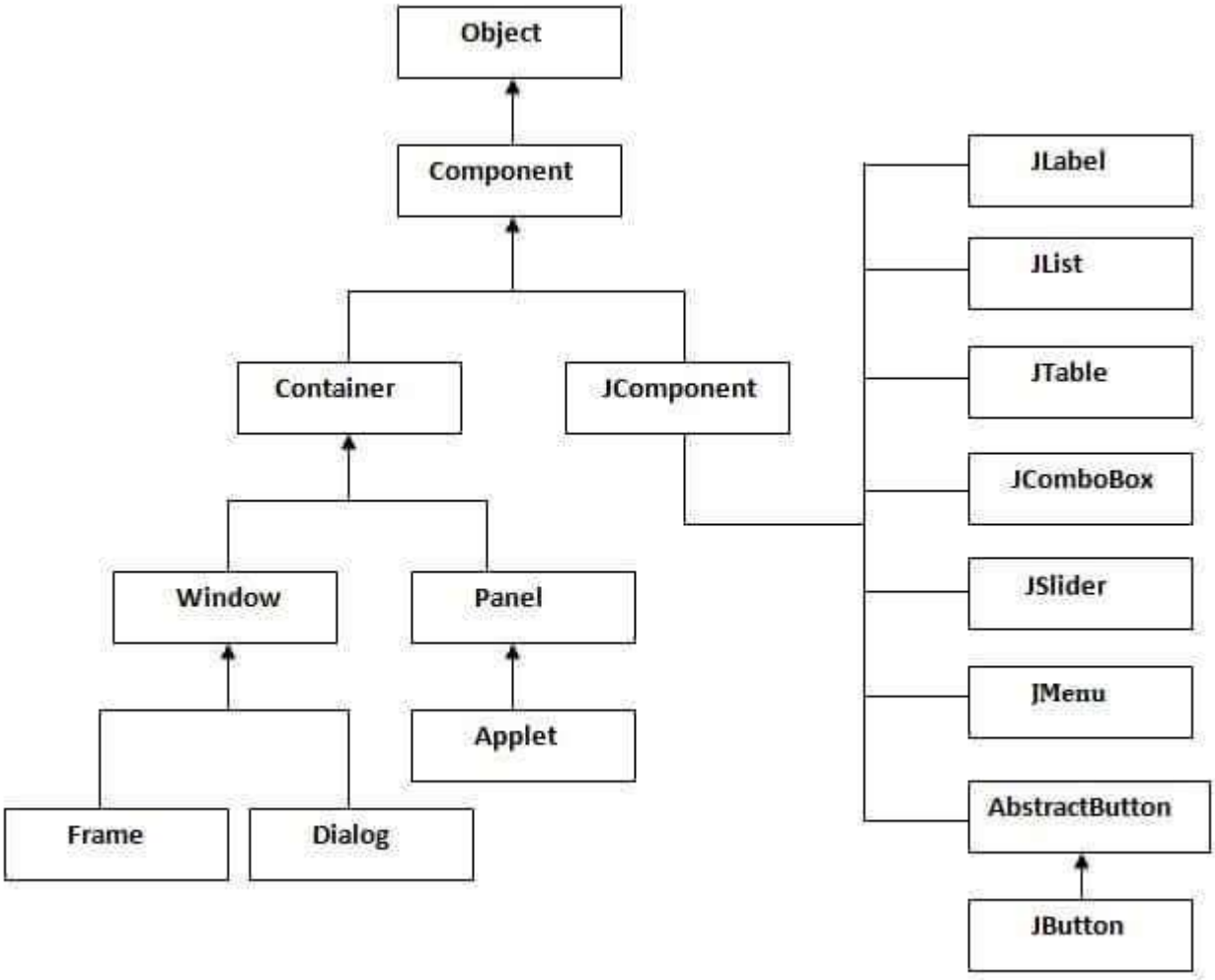
SWING

Java Swing is a part of Java Foundation Classes (JFC) that is used to create window-based applications. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Java Swing provides platform-independent and lightweight components.

The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Hierarchy of Java Swing classes



The methods of Component class are widely used in java swing that are given below.

public void add(Component c)	add a component on another component.
public void setSize(int width,int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.
public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

JFrame

The javax.swing.JFrame class is a type of container which inherits the java.awt.Frame class. JFrame works like the main window where components like labels, buttons, textfields are added to create a GUI.

Unlike Frame, JFrame has the option to hide or close the window with the help of setDefaultCloseOperation(int) method.

Constructors

JFrame()	It constructs a new frame that is initially invisible.
JFrame(GraphicsConfiguration gc)	It creates a Frame in the specified GraphicsConfiguration of a screen device and a blank title.
JFrame(String title)	It creates a new, initially invisible Frame with the specified title.
JFrame(String title, GraphicsConfiguration gc)	It creates a JFrame with the specified title and the specified GraphicsConfiguration of a screen

	device.

Methods

protected void	addImpl (Component comp, Object constraints, int index)	Adds the specified child Component.
protected JRootPane	createRootPane()	Called by the constructor methods to create the default rootPane.
protected void	frameInit()	Called by the constructors to init the JFrame properly.
void	setContentPane(Containe contentPane)	It sets the contentPane property
void	setIconImage(Image image)	It sets the image to be displayed as the icon for this window.
void	setJMenuBar(JMenuBar menubar)	It sets the menubar for this frame.
void	setLayeredPane(JLayeredPane layeredPane)	It sets the layeredPane property.
JRootPane	getRootPane()	It returns the rootPane object for this frame.
TransferHandler	getTransferHandler()	It gets the transferHandler property.

JDialog

The JDialog control represents a top level window with a border and a title used to take some form of input from the user. It inherits the Dialog class.

Unlike JFrame, it doesn't have maximize and minimize buttons.

Constructors

JDialog()	It is used to create a modeless dialog without a title and without a specified Frame owner.
JDialog (Frame owner)	It is used to create a modeless dialog with specified Frame as its owner and an empty title.
JDialog (Frame owner, String title, boolean modal)	It is used to create a dialog with the specified title, owner Frame and modality.

JPanel

The JPanel is a simplest container class. It provides space in which an application can attach any other component. It inherits the JComponents class.

It doesn't have title bar.

Constructors

JPanel()	It is used to create a new JPanel with a double buffer and a flow layout.
JPanel(boolean isDoubleBuffered)	It is used to create a new JPanel with FlowLayout and the specified buffering strategy.
JPanel(LayoutManager layout)	It is used to create a new JPanel with the specified layout manager.

JButton

The JButton class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed. It inherits AbstractButton class.

Constructors

JButton()	It creates a button with no text and icon.

JButton(String s)	It creates a button with the specified text.
JButton(Icon i)	It creates a button with the specified icon object.

Methods

void setText(String s)	It is used to set specified text on button
String getText()	It is used to return the text of the button.
void setEnabled(boolean b)	It is used to enable or disable the button.
void setIcon(Icon b)	It is used to set the specified Icon on the button.
Icon getIcon()	It is used to get the Icon of the button.
void setMnemonic(int a)	It is used to set the mnemonic on the button.
void addActionListener (ActionListener a)	It is used to add the action listener to this object.

JToggleButton

JToggleButton is used to create toggle button, it is two-states button to switch on or off.

Constructors

JToggleButton()	It creates an initially unselected toggle button without setting the text or image.
JToggleButton(Action a)	It creates a toggle button where properties are taken from the Action supplied.

JToggleButton(Icon icon)	It creates an initially unselected toggle button with the specified image but no text.
JToggleButton(Icon icon, boolean selected)	It creates a toggle button with the specified image and selection state, but no text.
JToggleButton(String text)	It creates an unselected toggle button with the specified text.
JToggleButton(String text, boolean selected)	It creates a toggle button with the specified text and selection state.
JToggleButton(String text, Icon icon)	It creates a toggle button that has the specified text and image, and that is initially unselected.
JToggleButton(String text, Icon icon, boolean selected)	It creates a toggle button with the specified text, image, and selection state.

Methods

AccessibleContext	getAccessibleContext()	It gets the AccessibleContext associated with this JToggleButton.
String	getUIClassID()	It returns a string that specifies the name of the l&f class that renders this component.
protected String	paramString()	It returns a string representation of this JToggleButton.
void	updateUI()	It resets the UI property to a value from the current look and feel.

JCheckBox

The JCheckBox class is used to create a checkbox. It is used to turn an option on (true) or off (false). Clicking on a CheckBox changes its state from "on" to "off" or from "off" to "on ".It inherits JToggleButton class.

Constructors

JCheckBox()	Creates an initially unselected check box button with no text, no icon.
JChechBox(String s)	Creates an initially unselected check box with text.
JCheckBox(String text, boolean selected)	Creates a check box with text and specifies whether or not it is initially selected.
JCheckBox(Action a)	Creates a check box where properties are taken from the Action supplied.

Methods

AccessibleContext getAccessibleContext()	It is used to get the AccessibleContext associated with this JCheckBox.
protected String paramString()	It returns a string representation of this JCheckBox.

JRadioButton

The JRadioButton class is used to create a radio button. It is used to choose one option from multiple options. It is widely used in exam systems or quiz.

It should be added in ButtonGroup to select one radio button only.

Constructors

JRadioButton()	Creates an unselected radio button with no text.
JRadioButton(String s)	Creates an unselected radio button with specified

	text.
JRadioButton(String s, boolean selected)	Creates a radio button with the specified text and selected status.

Methods

void setText(String s)	It is used to set specified text on button.
String getText()	It is used to return the text of the button.
void setEnabled(boolean b)	It is used to enable or disable the button.
void setIcon(Icon b)	It is used to set the specified Icon on the button.
Icon getIcon()	It is used to get the Icon of the button.
void setMnemonic(int a)	It is used to set the mnemonic on the button.
void addActionListener(ActionListener a)	It is used to add the action listener to this object.

JLabel

The object of JLabel class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by an application but a user cannot edit it directly. It inherits JComponent class.

Constructors

JLabel()	Creates a JLabel instance with no image and with an empty string for the title.
JLabel(String s)	Creates a JLabel instance with the specified text.
JLabel(Icon i)	Creates a JLabel instance with the specified image.

JLabel(String s, Icon i, int horizontalAlignment)	Creates a JLabel instance with the specified text, image, and horizontal alignment.

Methods

String getText()	t returns the text string that a label displays.
void setText(String text)	It defines the single line of text this component will display.
void setHorizontalAlignment (int alignment)	It sets the alignment of the label's contents along the X axis.
Icon getIcon()	It returns the graphic image that the label displays.
int getHorizontalAlignment()	It returns the alignment of the label's contents along the X axis.

JTextField

The object of a JTextField class is a text component that allows the editing of a single line text. It inherits JTextComponent class.

Constructors

JTextField()	Creates a new TextField
JTextField(String text)	Creates a new TextField initialized with the specified text.
JTextField(String text, int columns)	Creates a new TextField initialized with the specified text and columns.
JTextField(int columns)	Creates a new empty TextField with the specified number of columns.

Methods

void addActionListener(ActionListener l)	It is used to add the specified action listener to receive action events from this textfield.
Action getAction()	It returns the currently set Action for this ActionEvent source, or null if no Action is set.
void setFont(Font f)	It is used to set the current font.
void removeActionListener(ActionListener l)	It is used to remove the specified action listener so that it no longer receives action events from this textfield.

JTextArea

The object of a JTextArea class is a multi line region that displays text. It allows the editing of multiple line text. It inherits JTextComponent class

Constructors:

JTextArea()	Creates a text area that displays no text initially.
JTextArea(String s)	Creates a text area that displays specified text initially.
JTextArea(int row, int column)	Creates a text area with the specified number of rows and columns that displays no text initially.
JTextArea(String s, int row, int column)	Creates a text area with the specified number of rows and columns that displays specified text.

Methods:

void setRows(int rows)	It is used to set specified number of rows.
void setColumns(int cols)	It is used to set specified number of columns.
void setFont(Font f)	It is used to set the specified font.

void insert(String s, int position)	It is used to insert the specified text on the specified position.
void append(String s)	It is used to append the given text to the end of the document.

JList

The object of JList class represents a list of text items. The list of text items can be set up so that the user can choose either one item or multiple items. It inherits JComponent class.

Constructors:

JList()	Creates a JList with an empty, read-only, model.
JList(ary[] listData)	Creates a JList that displays the elements in the specified array.
JList(ListModel<ary> dataModel)	Creates a JList that displays elements from the specified, non-null, model.

Methods:

Void addListSelectionListener (ListSelectionListener listener)	It is used to add a listener to the list, to be notified each time a change to the selection occurs.
int getSelectedIndex()	It is used to return the smallest selected cell index.
ListModel getModel()	It is used to return the data model that holds a list of items displayed by the JList component.
void setListData(Object[] listData)	It is used to create a read-only ListModel from an array of objects.

JComboBox

The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits JComponent class.

Constructors:

JComboBox()	Creates a JComboBox with a default data model.
JComboBox(Object[] items)	Creates a JComboBox that contains the elements in the specified array.
JComboBox(Vector<?> items)	Creates a JComboBox that contains the elements in the specified Vector.

Methods:

void addItem(Object anObject)	It is used to add an item to the item list.
void removeItem(Object anObject)	It is used to delete an item to the item list.
void removeAllItems()	It is used to remove all the items from the list.
void setEditable(boolean b)	It is used to determine whether the JComboBox is editable.
void addActionListener (ActionListener a)	It is used to add the ActionListener.
void addItemListener(ItemListener i)	It is used to add the ItemListener.

JScrollPane

A JscrollPane is used to make scrollable view of a component. When screen size is limited, we use a scroll pane to display a large component or a component whose size can change dynamically.

Constructors

JScrollPane()	It creates a scroll pane. The Component parameter, when present, sets the scroll pane's client. The two int parameters, when present, set the vertical and horizontal scroll bar policies (respectively).
JScrollPane(Component)	
JScrollPane(int, int)	

JScrollPane(Component, int, int)	

Methods

void	setColumnHeaderView(Component)	It sets the column header for the scroll pane.
void	setRowHeaderView(Component)	It sets the row header for the scroll pane.
void	setCorner(String, Component)	It sets or gets the specified corner. The int parameter specifies which corner and must be one of the following constants defined in ScrollPaneConstants: UPPER_LEFT_CORNER, UPPER_RIGHT_CORNER, LOWER_LEFT_CORNER, LOWER_RIGHT_CORNER, LOWER_LEADING_CORNER, LOWER_TRAILING_CORNER, UPPER_LEADING_CORNER, UPPER_TRAILING_CORNER.
Component	getCorner(String)	
void	setViewportView(Component)	Set the scroll pane's client.

SUMMARY

Use Swing to create GUI.

ACTIVITIES

- Create a JFrame that displays a simple window with a label and a button. Clicking the button should display a message dialog.
- Create a JFrame with multiple JPanels, each using a different layout manager (e.g., BorderLayout, FlowLayout, GridLayout). Add buttons or other components to each panel to observe how layouts affect component positioning.

- Construct a JFrame that mimics a simple form (e.g., login screen or personal information form) using JLabels and JTextFields. Implement validation by changing text colors or displaying error messages.
- Develop a settings panel within a JFrame that includes checkboxes for enabling/disabling options and radio buttons for selecting options exclusively (like selecting a color theme).
- Create a JFrame with both a JComboBox and a JList. Populate them with data (e.g., items in a dropdown list or selectable items in a list). Implement event listeners to handle selection changes.
- Customize JButton and JToggleButton components by setting icons, tooltips, and background colors based on different states (like hover or click). Experiment with different look and feel options to change the appearance of components.
- Implement a JFrame with multiple buttons, each performing a different action (e.g., changing panel content, clearing form data). Use ActionListener to manage button clicks effectively.

SELF-ASSESSMENT QUESTIONS

1. What is Swing, and how does it differ from AWT?
2. How do you create a `JButton`? Provide an example of how to add it to a `JFrame`.
3. What is the purpose of `JLabel`, and how can you use it to display text and images?
4. How do you create a `JTextField`, and how can you retrieve user input from it?
5. What is the difference between `JTextArea` and `JTextField`?
6. How do you create a `JCheckBox`, and how can you determine its state?
7. Explain how to use a `JRadioButton` and group them using a `ButtonGroup`.
8. How do you create a `JComboBox` and add items to it?
9. What is the difference between `JList` and `JComboBox`?
10. What is a `JPanel`, and how can it be used to organize components?
11. How do you use layout managers like `FlowLayout`, `BorderLayout`, and GridLayout` in Swing?
12. How do you create and display a modal dialog using `JDialog`?
13. How do you attach an `ActionListener` to a button?
14. How can you create a custom component by extending `JComponent`?
15. Explain how to override the `paintComponent` method for custom drawing.