MODULE I

1.Explain OOP Concepts in Java. Explain the importance of Polymorphism, Encapsulation and Inheritance.

Object-Oriented Programming or OOPs refers to languages that uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

OOPs Concepts in Java includes:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object: It is a basic unit of Object-Oriented Programming and represents the real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of:

- **State:** It is represented by attributes of an object. It also reflects the properties of an object.
- **Behaviour:** It is represented by methods of an object. It also reflects the response of an object with other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Example of an object: A dog is an object because it has states like color, name, breed, etc. as well as behaviours like wagging the tail, barking, eating, etc.

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. In general, class declarations can include these components, in order:

1. **Modifiers**: A class can be public or has default access.
2. **Class name:** The name should begin with an initial letter capitalized by convention.
3. **Superclass (if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. **Interfaces (if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. **Body:** The class body surrounded by braces, { }.

Example:

```
public class Dog {
String name; int age; String color;
void barking() {
}
void eating() {
} }
```

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. The keyword used for inheritance is **extends**. Important terminologies in inheritance include:

- **Super Class:** The class whose features are inherited is known as superclass (or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as subclass (or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Example:

```
class Color extends Dog {
void eating(){
} }
```

Polymorphism: If one task is performed in different ways, it is known as polymorphism. Polymorphism refers to the ability of OOPs programming languages to differentiate between entities with the same name efficiently. For example: to draw something, for example, shape, triangle, rectangle, etc. In Java, we use method overloading and method overriding to achieve polymorphism.

Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word "poly" means many and "morphs" means forms, so it means many forms.

In Java polymorphism is mainly divided into two types:

- **Compile time polymorphism**: It is also known as static polymorphism. This type of polymorphism is achieved by **method overloading** or operator overloading.

Operator Overloading: Java also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So, a single operator '+' when placed between integer operands adds them and when placed between string operands concatenates them.

- **Runtime polymorphism**: It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by **method overriding**.

Abstraction: Hiding internal details and showing functionality is known as abstraction. For example, by phone call, we don't know the internal processing. In java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

Encapsulation: Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield. A java class is the example of encapsulation.

- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.
- As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.
- Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

2.What are the features of Java?

Features of Java

The features of Java are also known as *java* buzzwords.

Most important features of Java are given below:

- Simple
- Object-Oriented
- Portable
- Platform independent
- Secured
- Robust
- Architecture neutral
- Interpreted
- High Performance
- Multithreaded
- Distributed
- Dynamic

Simple
Java is very easy to learn, and its syntax is simple, clean and easy to understand.

Object-oriented
Java is an object-oriented programming language. Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Platform Independent
Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. Java platform software environment in which a program runs. It has two components:

- Runtime Environment
- API (Application Programming Interface)

Java code can be run on multiple platforms, for example, Windows, Linux, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere (WORA).

Secured
Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- **No explicit pointer**
- **Java Programs run inside a virtual machine sandbox**
- **Classloader:** Classloader in Java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Robust
Robust simply means strong. Java is robust because:

- It uses strong memory management.
- There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java.

Architecture-neutral
Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

Portable
Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

High-performance
Java is faster than other traditional interpreted programming languages because Java bytecode is close to native code.

Distributed
Java is distributed because it facilitates users to create distributed applications in Java. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

Multi-threaded
A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

Dynamic
Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++. Java supports dynamic compilation and automatic memory management (garbage collection).

3.Explain JVM with neat diagram.

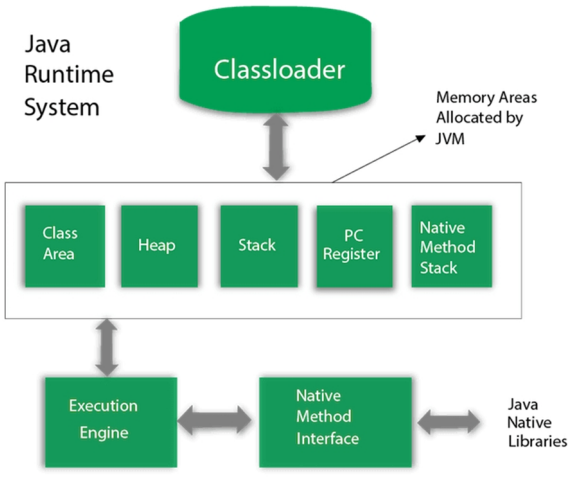
JVM (Java Virtual Machine)

- JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.
- JVMs are available for many hardware and software platforms. i.e. JVM is platform dependent.
- Its implementation is known as JRE (Java Runtime Environment).
- Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

Operations performed by JVM

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM Architecture



Classloader : Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.

- **Bootstrap ClassLoader**
- **Extension ClassLoader**
- **System/Application ClassLoader**

Class(Method) Area : Class Area stores per-class structures such as the code for methods.

Heap : It is the runtime data area in which objects are allocated.

Stack : Java Stack stores frames.

Program Counter Register : PC register contains the address of the Java virtual machine instruction currently being executed.

Native Method Stack : It contains all the native methods used in the application.

Execution Engine : It contains:

- **A virtual processor**
- **Interpreter:** Read bytecode stream then execute the instructions.
- **Just-In-Time (JIT) compiler:** It is used to reduce the amount of time needed for compilation.

Java Native Interface: It is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc.

4.Explain the structure of Java program with example.

A Java program involves the following sections:

- Documentation Section
- Package Statement
- Import Statements
- Interface Statement
- Class Definition
- Main Method Class

Documentation
You can write a comment in this section. Comments are beneficial for the programmer because they help them understand the code. These are optional, but we suggest you use them because they are useful to understand the operation of the program, so you must write comments within the program.
/ comment_section */*

Package Statement
You can create a package with any name. A package is a group of classes that are defined by a name. That is, if you want to declare many classes within one element, then you can declare it within a package. It is an optional part of the program, i.e., if you do not want to declare any package, then there will be no problem with it, and you will not get any errors. Here, the package is a keyword that tells the compiler that package has been created.
Syntax : package package_name;

Import Statement
This line indicates that if you want to use a class of another package, then you can do this by importing it directly into your program.
Syntax : import java.io.;*

Interface Statement
Interfaces are like a class that includes a group of method declarations. It's an optional section and can be used when programmers want to implement multiple inheritances within a program.
Syntax : Interface interface_name { }

Class Definition
A Java program may contain several class definitions. Classes are the main and essential elements of any Java program.
Syntax : class class_name { }

Main Method Class
Every Java stand-alone program requires the main method as the starting point of the program. This is an essential part of a Java program. There may be many classes in a Java program, and only one class defines the main method. Methods contain data type declaration and executable statements.

Example of a Simple Java program

```
//Name of this file will be "Hello.java"
public class Hello
{
  /* Writes the word "Welcome" on the screen */
  public static void main(String[] args)
  {
    System.out.println("Welcome");
  }
}
```

5.What is lexical structure of Java?

The **lexical structure** of a programming language is a set of basic rules that specify how you write programs in this particular language.

A source code of a Java program consists of **tokens**. Tokens are atomic code elements. In Java we have comments, identifiers, literals, operators, separators, and keywords.

Java programs are composed of characters from the Unicode character set.

1.Comments

Comments are used by humans to clarify source code. There are three types of comments in Java.

- **Single-line Comments** - //comment
- **Multi-line Comments** - /* comment */
- **Documentation Comments** - /** documentation */

2.Java white space

White space in Java is used to separate tokens in the source file. It is also used to improve readability of the source code. The amount of space put between tokens is irrelevant for the Java compiler. The white space should be used consistently in Java source code. Eg: int<space>i;

3.Java identifiers

Identifiers are names for variables, methods, classes, or parameters. Identifiers can have alphanumerical characters, underscores and dollar signs ($). It is an error to begin a variable name with a number.

White space in names is not permitted.

Identifiers are case sensitive. This means that Name, name, or NAME refer to three different variables. Identifiers also cannot match language keywords.

There are also conventions related to naming of identifiers. The names should be descriptive. We should not use cryptic names for our identifiers. If the name consists of multiple words, each subsequent word is capitalized.

Eg:

- String name23; int _col; short car_age; //valid
- String 23name; int %col; short car age; //invalid

4.Java literals

A literal is a textual representation of a particular value of a type. Literal types include Boolean, integer, floating point, string, null, or character. Technically, a literal will be assigned a value at compile time, while a variable will be assigned at runtime.

int age = 29;

String nationality = "Hungarian";

Here we assign two literals to variables. 29 and Hungarian are literals.

5.Java operators

An operator is a symbol used to perform an action on some value. Operators are used in expressions to describe operations involving one or more operands. There are different types of operators in Java.

- **Arithmetic Operators**: Arithmetic operators are used for mathematical expressions.

+	Addition (unary plus)
-	Subtraction (unary minus)
*****	Multiplication
/	Division
%	Modulus
++	Increment

- **Bitwise Operators:** Bitwise operator works on bits and performs bit-by-bit operation.

~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
<<	Shift left

- **Relational Operator:** Relational operator operates on relationship that one operand has to the other.

==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

- **Logical Operator:** Boolean Logical operator operate only on Boolean operands.

&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT

- **Assignment Operators:**

=	Simple Assignment Operator
-=	Subtract and Assignment Operator
+=	Add and Assignment Operator
/=	Divide and Assignment Operator
***=**	Multiply and Assignment Operator
%=	Modulus and Assignment Operator
>>=	Right Shift and Assignment Operator
<<=	Left Shift and Assignment Operator
^=	Bitwise Exclusive OR and Assignment Operator
&=	Bitwise and Assignment Operator
|=	Bitwise Inclusive OR and Assignment Operator

Java separators

A separator is a sequence of one or more characters. Following are the separators in Java:

( )	Parentheses	Used to contain the lists of parameters in method definition and invocation. Also used for defining the precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contains the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates the statements
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement
.	Period	Used to separate packages names from sub packages and classes. Also used to separate a variable or method from a reference variable.
::	Colons	Used to create a method or constructor reference

Java keywords

A keyword is a reserved word in Java language. Keywords are used to perform a specific task in the computer program. For example, to define variables, do repetitive tasks or perform logical operations.

Java is rich in keywords. Some are:

abstract	continue	for	new	switch
assert	default	goto	package	while
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	float	return	const
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	super	volatile

Java conventions

Conventions are best practices followed by programmers when writing source code. Each language can have its own set of conventions. Conventions are not strict rules; they are merely recommendations for writing good quality code. Some of the conventions that are recognized by Java programmers are:

- Class names begin with an uppercase letter.
- Method names begin with a lowercase letter.
- The public keyword precedes the static keyword when both are used.
- The parameter name of the main() method is called args.
- Constants are written in uppercase.
- Each subsequent word in an identifier name begins with a capital letter.

6.Explain Java variables. What is Dynamic initialization of Variables? What is the scope and lifetime of Java variables?

Variables in Java

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

Java Variable Declaration

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

```
type identifier;
or,
type identifier = value;
or,
type identifier1, identifier2;
```

There are three types of variables in Java:

Instance Variables	A variable which is declared inside a class and outside all the methods and blocks is an instance variable.
Class Variables	A variable which is declared inside a class, outside all the blocks and is marked static is known as a class variable.
Local Variables	All other variables which are not instance and class variables are treated as local variables including the parameters in a method.

```
class Sample
{
        int x,y;      //instance variables
        static int result;     //class variable
        void add(int a, int b)     //local variables
        {
                x=a;
                y=b;
                int sum=x+y;     //sum is a local variable
                System.out.println("Sum is "+sum);
        }
        public static void main(String args[])
        {
                Sample obj= new Sample();
                obj.add(10,20);
        }
}
```

Dynamic Initialization in Java

Apart from using constants as initializer, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared. For example, following is a short program that computes the length of the hypotenuse of a right-angle triangle given the lengths of its two opposing sides:

```
public class JavaProgram
{
   public static void main(String args[])
   {

      double a = 3.0, b = 4.0;

      /* c is dynamically initialized */
      double c = Math.sqrt(a * a + b * b);
      System.out.println("Hypotenuse is " + c);

   }
}
```

Scope and Lifetime of Java Variables

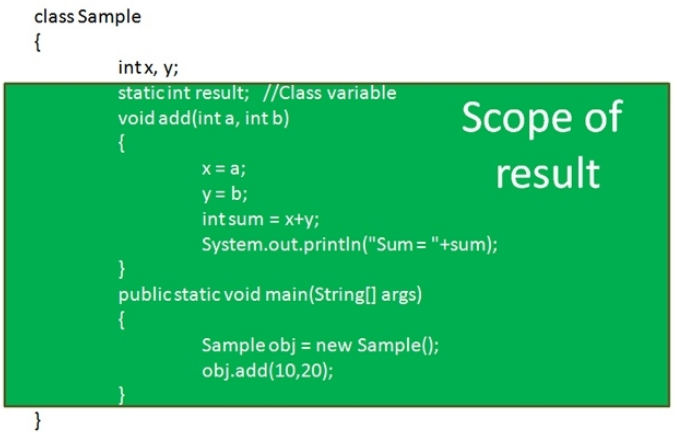
Scope of a variable refers to in which areas or sections of a program can the variable be accessed, and **lifetime** of a variable refers to how long the variable stays alive in memory. General convention for a variable's scope is, it is accessible only within the block in which it is declared. A block begins with a left curly brace '{' and ends with a right curly brace '}'.

- **Scope of an instance variable** is throughout the class except in static methods. **Lifetime of an instance variable** is until the object stays in memory.

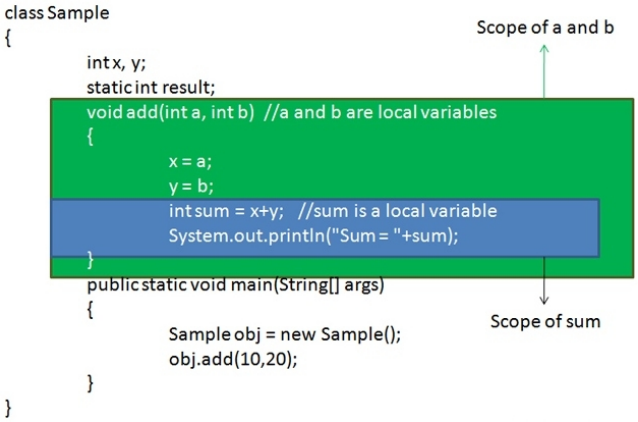
```
class Sample
{
    int x, y;   //instance variables
    static int result;
    void add(int a, int b)  //a and b are local variables
    {
            x = a;
            y = b;
            int sum = x+y;   //Sum
            System.out.println("Sum = "+sum);
    }
    public static void main(String[] args)
    {
            Sample obj = new Sample();
            obj.add(10,20);
    }

}
```

Scope of x and y

- **Scope of a class variable** is throughout the class. **Lifetime of a class variable** is until the end of the program or as long as the class is loaded in memory.



- **Scope of a local variable** is within the block in which it is declared. **Lifetime of a local variable** is until the control leaves the block in which it is declared.



7.What are data types Java?

Java Data Types

A variable in Java must be a specified data type.

Data types are divided into two groups:

- Primitive data types
- Non-primitive data types

Primitive Data Types

A primitive data type specifies the size and type of variable values, and it has no additional methods.

There are eight primitive data types in Java:

Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

Non-Primitive Data Types

Non-primitive data types are called **reference types** because they refer to objects.

The main difference between **primitive** and **non-primitive** data types are:

- Primitive types are predefined in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String).
- Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
- A primitive type has always a value, while non-primitive types can be null.

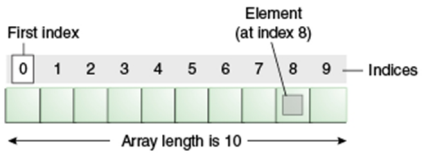
- A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.
- The size of a primitive type depends on the data type, while non-primitive types have all the same size.
- Examples of non-primitive types are Strings, Arrays, Classes, Interface, etc.

8.What are Arrays? Explain the declaration of array variables. What are strings? Explain string manipulation functions in Java.

Java Arrays

- **Array** is an object which contains elements of a similar data type.
- The elements of an array are stored in a contiguous memory location.
- It is a data structure where we store similar elements.
- We can store only a fixed set of elements in a Java array.
- We can store primitive values or objects in an array in Java.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.



Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages

- **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

There are two types of array.

1. Single Dimensional Array
2. Multidimensional Array

Single Dimensional Array

Syntax:
dataType[] arr; (or)
dataType []arr; (or)
dataType arr[];

Declaration, Instantiation and Initialization of an Array
We can declare, instantiate and initialize the java array together by:

```
int a[]={33,3,4,5};     //declaration, instantiation and initialization
```

Passing Array to a Method in Java
We can pass the java array to method

```
class Test {
static void printArray(int arr[]){
---
}
```

Pointer to an Array
Syntax:

```
data_type (*var_name)[size_of_array];
```

Anonymous Array in Java

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

```
public class Test {
static void printArray(int arr[]){
---
}
public static void main(String args[]){
printArray(new int[]{10,22,44,66});   //passing anonymous array to method
}}
```

Multidimensional Array in Java

In such case, data is stored in row and column-based index (also known as matrix form).

instantiating Multidimensional Array in Java
int[][] arr=new int[2][2]; //3 row and 3 column

initializing Multidimensional Array in Java
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;

Java Strings

In Java, **string** is basically an object that represents sequence of char values.
An array of characters works same as Java string.

For example:
char[] ch={'t','e','s','t'};
String s=new String(ch);
O/P = test

The java.lang.String class provides a lot of methods to perform operations on strings. These are called String manipulation functions or **accessor methods** of String class. Methods used to obtain information about an object are known as accessor methods.

Some of the string manipulation functions are:

concat()

- used for concatenating or joining two or more strings.
- using **concat()** method of String class or by using arithmetic **"+"** operator.

```
String str3 = str1.concat(str2);
      Or
String str3 = str1 + str2;
```

length()

- returns the number of characters contained in the string object.

```
Str1.length()
```

indexOf()

- used to determine the location of a specific character that you specify.

```
str1.indexOf('S')
```

charAt()

- used to get character at a specific location

```
str1.charAt(5)
```

contains()

- returns either **true** or **false** by checking if the string contains the specified sequence of char values

```
str_Sample.contains("tar")
```

tolowercase() & **touppercase()**

- converts the string either to lowercase or uppercase

```
str1.toLowerCase()
str1.toUpperCase()
```

MODULE II

1.What are control statements in Java?

Control statements in Java

Control statements are used to control the flow of execution of program based on certain conditions.

There are three types of control statements:

- Conditional Control Statements
- Looping Control Statements
- Unconditional Control Statements/Jump Statements

Conditional Control Statements

Conditional Control Statements allows the program to select between the alternatives during the program execution.
They are also called as decision-making statements or selection statements.

1. If statement
It will go inside the block only if the condition is true otherwise, it will not execute the block.

```
if(condition){
// statements (if Block)
}
//other statements
```

2. If-Else Statement
If the condition is true then, it will execute the If block. Otherwise, it will execute the Else block.

```
if(condition){
// statements (if Block)
}
else{
// statements (Else block)
}
//other statements
```

3. If Else-If statement
If the condition is true, then it will execute the If block. Otherwise, it will execute the Else-If block. Again, if the condition is not met, then it will move to the else block.

```
if(condition){
// statements (if Block)
}
else if{
// statements (Else-If block)
}
else{
//statements(Else Block)
}//other statements
```

4. Switch Statement
Switch statement allows program to select one action among multiple actions during the program execution.

```
Switch(variable/value/expression) {
Case :
// statements [];
Case :
// statements [];
…
default:
// statements [];
}
```

- Based on the argument in the switch statement suitable case value will be selected and executed.
- If no matching case found, then the default will be executed.
- It is optional to write a break statement at the end of each case statement.

Looping Control Statements

These are used to execute a block of statements multiple times. It means it executes the same code multiple times so it saves code. These are also called Iteration statements.

There are three types of looping control statements:

- For loop
- While loop
- Do-while loop

1. For loop
It executes the code until condition is false.
It is used when number of iterations are known.

```
for(initialization; condition; increment/decrement){
//statements (For Body)
}
```

2. While loop
While loop executes till the condition becomes false.

```
while(condition){
// statements
}
```

3. Do-while loop

- When you are using for or while, then it will execute the loop body only if the condition if true.
- In do-while loop, it will execute the loop first, then it checks the condition. So, it will execute the loop at least once.
- It is called exit-controlled loop while for & while loop are called entry-controlled loop.

```
do{
// statements
}while(condition);
```

Unconditional Control Statements/Jump Statements

1. break Statement

It is used within any control statements. It is used to terminate the execution of the current loop or switch statements.
Syntax: **break;**

2. continue Statement

It is used to continue the execution of the current loop with the next iteration.
Syntax: **continue;**

2.What are methods in Java?

Method

A method is a collection of statements that perform some specific task and return result to the caller. A method can perform some specific task without returning anything. Methods allow us to **reuse** the code without retyping the code. In Java, every method must be part of some class.

Method Declaration

In general, method declaration has six components:

- **Access Modifier**: Defines **access type** of the method i.e. from where it can be accessed in your application. In Java, there 4 type of the access specifiers.
  - **public:** accessible in all class in your application.
  - **protected:** accessible within the package in which it is defined and in its subclasses (including subclasses declared outside the package).
  - **private:** accessible only within the class in which it is defined.
  - **default (declared/defined without using any modifier):** accessible within same class and package within which its class is defined.
- **The return type**: The data type of the value returned by the method or void if does not return a value.
- **Method Name**: the rules for field names apply to method names as well, but the convention is a little different.
- **Parameter list**: Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().
- **Exception list**: The exceptions you expect by the method can throw, you can specify these exception(s).
- **Method body**: it is enclosed between braces. The code you need to be executed to perform your intended operations.

Eg:
public int max(int x, int y) { //method body }

Method signature: It consists of the method name and a parameter list(without return type). Eg: max(int x, int y)

3.Can a class be static in Java?

Yes, we can have static class in java. In java, we have static instance variables as well as static methods and also static block. Classes can also be made static in Java.
But,
Java allows us to define a class within another class. Such a class is called a **nested class**. The class which enclosed nested class is known as **Outer class**. In java, we can't make Outer class static. ***Only nested classes can be static***.

Eg:

```
class Outercls{
  private static String msg = "Welcome";  //should be static
  public static class Nestedcls {     //static class
     public void printMessage() {
       System.out.println("Message from nested static class: " + msg);
     }
}

}
```

4.What is constructor? Explain different types of constructors with examples. Can a class have different constructor? (Constructor Overloading)

Constructors in Java

- In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an object is created using the new() keyword, at least one constructor is called.
- It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.
- We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

There are three rules defined for the constructor.

1. Constructor name must be the same as its class name.
2. A Constructor must have no explicit return type.
3. A Java constructor cannot be abstract, static, final, and synchronized.

Types of Java constructors

There are two types of constructors in Java:

- Default constructor
- Parameterized constructor

Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Eg:

```
class Test{
Test(){   //default constructor
System.out.println("Success");
}
```

Java Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

Eg:

```
class Test{
int id;
String name;
Test(int i, String n){   //parameterized constructor
id = i;
name = n;
}
```

Constructor Overloading in Java

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

```
class Test{
int id;
String name;
int age;
Test(int i, String n){    //creating two arg constructor
id = i;
name = n;
}
Test(int i, String n, int a){    //creating three arg constructor
id = i;
name = n;
age=a;
}
void display(){
System.out.println(id+" "+name+" "+age);
}
public static void main(String args[]){
Test t1 = new Test(101,"John");
Test t2 = new Test(102,"Jerry",20);
t1.display();
t2.display();
} }
```

6.Explain method overloading and method overriding in Java with suitable examples. Distinguish between method overloading and method overriding.

Method Overloading

Method Overloading is a **Compile time polymorphism**. In method overloading, more than one method shares the same method name with different method signature(parameters) in the class. In method overloading, return type can or cannot be the same, but we must have to change the parameter because in java, we cannot achieve the method overloading by changing only the return type of the method.

Example of Overloading

```
class Test{
static int add(int a, int b){
return a+b;}
static int add(int a, int b, int c){
return a+b+c;}
public static void main(String args[]) {
System.out.println(add(4, 6));
System.out.println(add(4, 6, 7));
} }
```

O/P :
10
17

Method Overriding

Method Overriding is a **Run time polymorphism**. In method overriding, derived class provides the specific implementation of the method that is already provided by the base class or parent class. In method overriding, return type must be same or co-variant (return type may vary in same direction as the derived class).

Example of Overriding

```
class Animal{
void eat(){System.out.println("eating.");}
}
class Dog extends Animal{
void eat(){System.out.println("Dog is eating.");}
}
class Test{
public static void main(String args[]) {
Dog d1=new Dog();
Animal a1=new Animal();
d1.eat();
a1.eat();
} }
```

O/P :
Dog is eating
eating

Difference between Method Overloading and Method Overriding in Java:

S.NO	METHOD OVERLOADING	METHOD OVERRIDING
1.	Method overloading is a compile time polymorphism.	Method overriding is a run time polymorphism.
2.	It helps to rise the readability of the program.	While it is used to grant the specific implementation of the method which is already provided by its parent class or super class.
3.	It occurs within the class.	It is performed in two classes with inheritance relationship.
4.	Method overloading may or may not require inheritance.	Method overriding always needs inheritance.
5.	In this, methods must have same name and different signature.	In this, methods must have same name and same signature.
6.	In method overloading, return type can or cannot be the same, but we must have to change the parameter.	While in this, return type must be same or co-variant.

7.What is inheritance in Java? Explain different types of inheritance.

Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviours of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

inheritance in java is mainly used:

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Sub Class: Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

Super Class: Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.

The syntax of Java Inheritance:

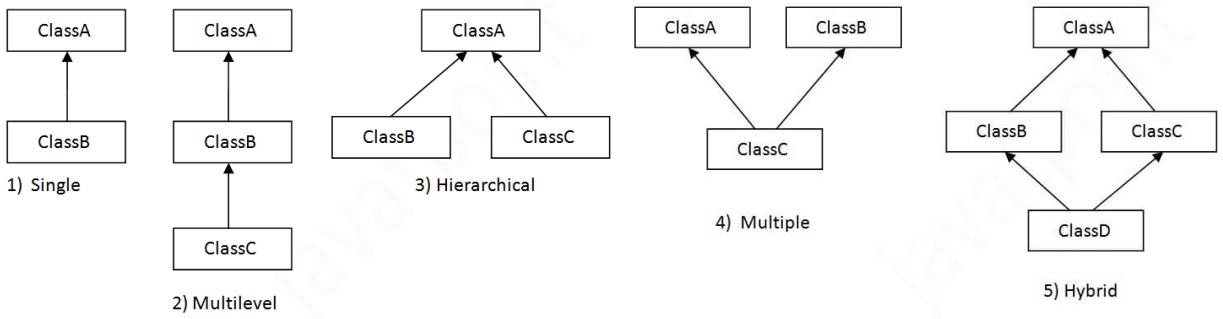
```
class subclass_name extends superclass_name
{
  //methods and fields
}
```

Here, the **extends** keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

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



Single Inheritance Example

When a class inherits another class, it is known as a single inheritance.

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

O/P
barking...
eating...

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as multilevel inheritance.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

O/P
weeping...
barking...
eating...

Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as hierarchical inheritance.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

O/P
meowing...
eating...

8.What are interfaces in Java? How multiple inheritance can be implemented in Java?

Interface in Java

An **interface** in Java is a blueprint of a class. It has static constants and abstract methods.

- The interface in Java is a mechanism to achieve abstraction.
- There can be only abstract methods in the Java interface, not method body.
- It is used to achieve multiple inheritance in Java.
- An interface is declared by using the interface keyword.
- It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default.
- A class that implements an interface must implement all the methods declared in the interface.

Syntax:
interface <interface_name> {
}

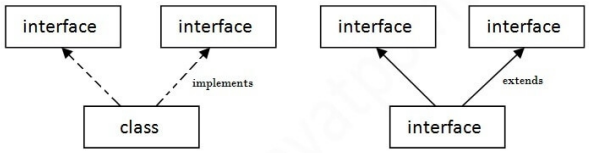
Java Interface Example

```
interface Printable{
void print();
}
class Test implements Printable{
public void print(){
System.out.println("Hello");
}
public static void main(String args[]){
Test obj = new Test();
obj.print();
}
}
```

Output:
Hello

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class Test implements Printable,Showable{
public void print(){
System.out.println("Hello");
}
public void show(){
System.out.println("Welcome");
}
public static void main(String args[]){
Test obj = new Test();
obj.print();
obj.show();
}
}
```

Output:
Hello
Welcome

9.What are abstract classes and methods in Java? Explain its uses. Write a program illustrating use of abstract classes and methods.

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.
There are two ways to achieve abstraction in java:

- Abstract class (0 to 100%)
- Interface (100%)

Abstract class in Java

- A class which is declared as abstract is known as an **abstract class**.
- They are used to provide some common functionality across a set of related classes while also allowing default method implementations.
- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It can have constructors and static methods also.
- It needs to be extended and its method implemented.
- It can have final methods which will force the subclass not to change the body of the method.
- It cannot be instantiated.

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of Abstract class that has an abstract method

Here, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Vehicle class.

```
abstract class Bike{    //abstract class
abstract void run();   //abstract method
}
class Vehicle extends Bike{
void run(){
System.out.println("running safely");
}
public static void main(String args[]){
Bike obj = new Vehicle();
obj.run();
}
}
```

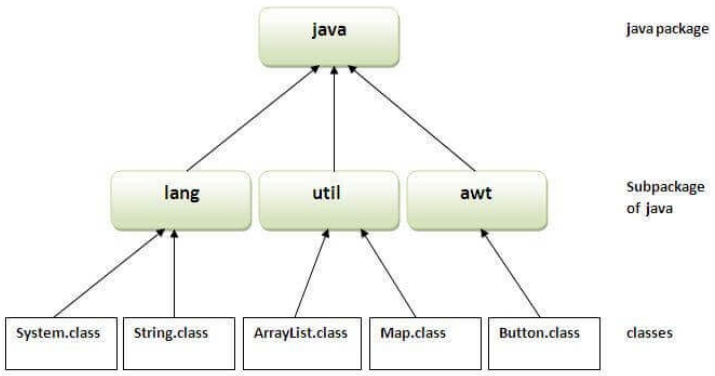
Output:
running safely

10.Explain about packages in Java.

Java Package

A java package is a group of similar types of classes, interfaces and sub-packages.

- Package in java can be categorized in two form, built-in package and user-defined package.
- There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
- Advantage of Java Package
  1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
  2) Java package provides access protection.
  3) Java package removes naming collision.



Simple example of java package

The package keyword is used to create a package in java.

```
package mypack;
public class Simple{
public static void main(String args[]){
System.out.println("Welcome to package");
} }
```

Compiling a java package program

javac -d directory <javafilename>
For example:
javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. If you want to keep the package within the same directory, you can use . (dot).

Running a java package program

Java <packagename>.<javafilename>
For example:
java mypack.Simple

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using package.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.
The import keyword is used to make the classes and interface of another package accessible to the current package.

```
//save by A.java
package pack;
public class A{
public void msg(){
System.out.println("Hello");
} }
```

```
//save by B.java
package mypack;
import pack.*;
class B{
public static void main(String args[]){
A obj = new A();
obj.msg();
} }
```

Output:Hello

2) Using package.classname

If you import package.classname then only declared class of this package will be accessible.

```
//save by A.java
package pack;
public class A{
public void msg(){System.out.println("Hello");
} }
```

```
//save by B.java
package mypack;
import pack.A;
class B{
public static void main(String args[]){
A obj = new A();
obj.msg();
} }
```

Output:Hello

3) Using fully qualified name

If you use fully qualified name, then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

```
//save by A.java
package pack;
public class A{
public void msg(){
System.out.println("Hello");
} }

//save by B.java
package mypack;
class B{
public static void main(String args[]){
pack.A obj = new pack.A();   //using fully qualified name
obj.msg();
}
}
```

Output:Hello

11.What is final variable in Java?

Final variables in Java

In Java, when **final** keyword is used with a variable of primitive data types (int, float, etc.), value of the variable cannot be changed.

For example, following program gives error because i is final.

```
public class Test {
public static void main(String args[]) {
final int i = 10;
i = 30; // Error because i is final.
}
}
```

MODULE III

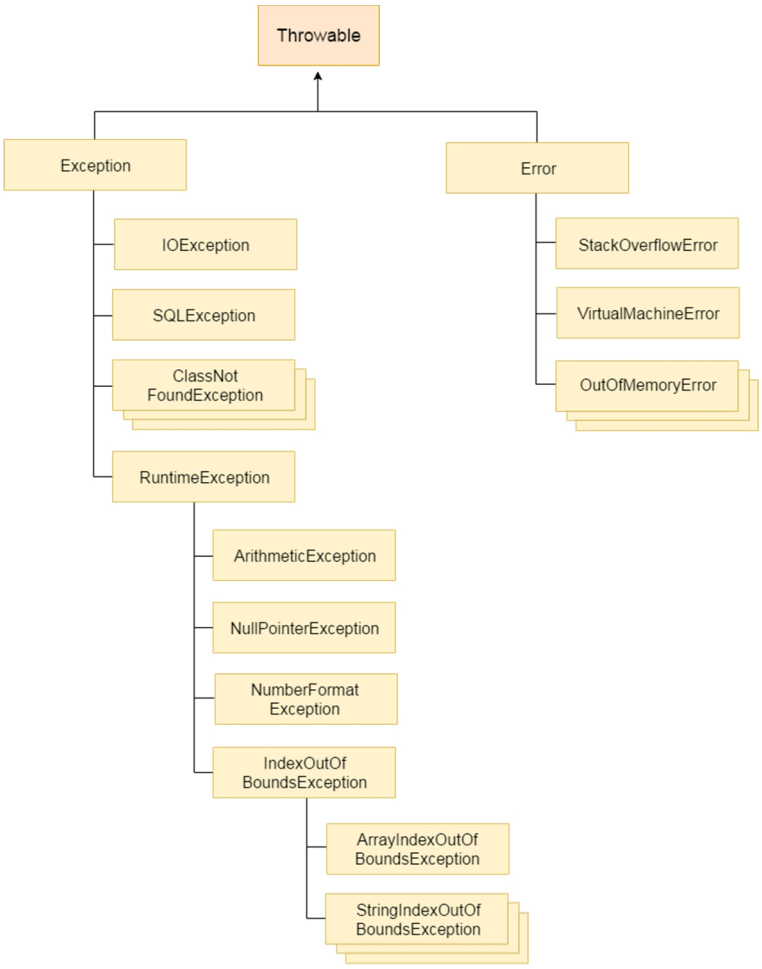
1.What is Exception Hierarchy? What are the different types of Exceptions?

Exceptions in Java

An exception is an unwanted or unexpected event, which occurs during the execution of a program. In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Hierarchy of Java Exception class

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error. A hierarchy of Java Exception classes are given below:



Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

- Checked Exception
- Unchecked Exception
- Error

1) Checked Exception

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

An Error indicates serious problem that a reasonable application should not try to catch. Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

2.Explain Exception handling in Java. Explain the uses of keywords try, catch and finally with examples. What is the difference between throw and throws?

Exception Handling in Java

The **Exception Handling** in Java is one of the powerful mechanisms *to handle the runtime errors* so that normal flow of the application can be maintained.

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling. Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception. It is used in the method body to throw an exception. **"throw"** is followed by an instance of Exception class. Eg: throw new ArithmeticException("Arithmetic Exception");
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature. **"throws"** is followed by Exception class names. Eg: throws ArithmeticException;

Default Exception Handling

Whenever inside a method, if an exception has occurred, the method creates an Object known as Exception Object and hands it off to the run-time system **(JVM).** The exception object contains name and description of the exception, and current state of the program where exception has occurred. Creating the Exception Object and handling it to the run-time system is called throwing an Exception. There might be the list of the methods that had been called to get to the method where exception was occurred. This ordered list of the methods is called **Call Stack**. Now the following procedure will happen.

- The run-time system searches the call stack to find the method that contains block of code that can handle the occurred exception. The block of the code is called **Exception handler**.
- The run-time system starts searching from the method in which exception occurred, proceeds through call stack in the reverse order in which methods were called.
- If it finds appropriate handler, then it passes the occurred exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
- If run-time system searches all the methods on call stack and couldn't have found the appropriate handler then run-time system handover the Exception Object to **default exception handler,** which is part of run-time system. This handler prints the exception information in the following format and terminates program abnormally.

```
// Java program to demonstrate how exception is thrown.
class ThrowsExecp
{
   public static void main(String args[]){
   String str = null;
   System.out.println(str.length());
  }
}
Output :
Exception in thread "main" java.lang.NullPointerException at ThrowsExecp.main
```

Structure of Exception Handling

```
try {
// block of code to monitor for errors
}
catch (<Exception type1> <Exception object>) {
// exception handler for Exception type1
}
catch (<Exception type2> <Exception object>) {
// exception handler for Exception type2
}
finally {    // optional
// block of code to be executed after try block ends
}
```

Important points

- In a method, there can be more than one statements that might throw exception, so put all these statements within its own **try** block and provide separate exception handler within **catch** block for each of them.
- If an exception occurs within the **try** block, that exception is handled by the exception handler associated with it. To associate exception handler, we must put **catch** block after it. There can be more than one exception handlers. Each **catch** block is an exception handler that handles the exception of the type indicated by its argument. The argument, ExceptionType declares the type of the exception that it can handle and must be the name of the class that inherits from **Throwable** class.
- For each try block there can be zero or more catch blocks, but **only one** finally block.
- The finally block is optional. It always gets executed whether an exception occurred in try block or not. If exception occurs, then it will be executed after **try and catch blocks.** And if exception does not occur then it will be executed after the **try** block. The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.

Exception Handling example

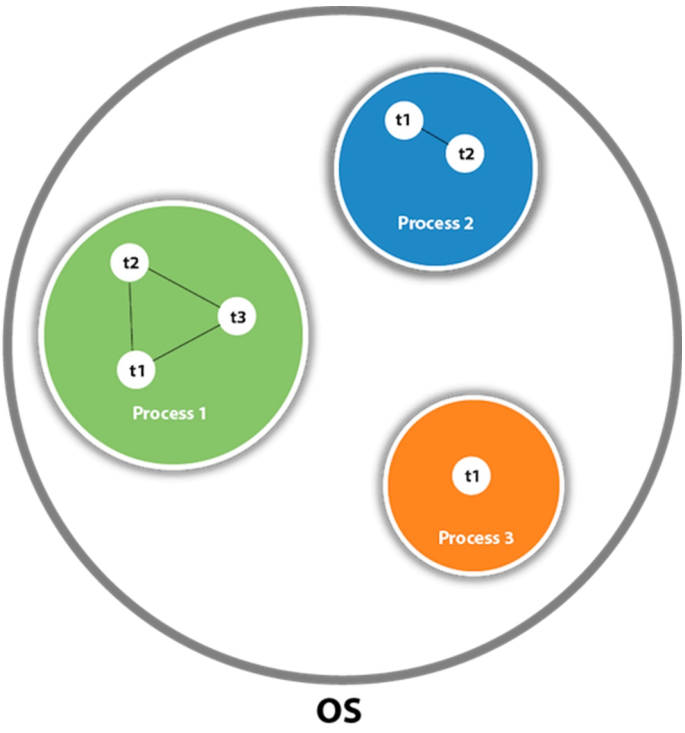
```
public class JavaExceptionExample{
public static void main(String args[]){
try{
//code that may raise exception
int data=100/0;
}
catch(ArithmeticException e){
System.out.println(e);
}
//rest code of the program
finally {    //optional
System.out.println("rest of the code.");
} }
}
```

Output:
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code.

3.What are threads in Java? Explain multithreading. Explain thread life cycle.

Thread

A thread is a lightweight sub-process, the smallest unit of processing. All Java programs have at least one thread, known as the main thread, which is created by the Java Virtual Machine (JVM) at the program's start, when the main() method is invoked with the main thread.
In Java, creating a thread is accomplished by implementing an interface and extending a class. Every Java thread is created and controlled by the java.lang.Thread class.



Multithreading in Java

- **Multithreading in Java** is a process of executing multiple threads simultaneously.
- Multiprocessing and multithreading, both are used to achieve multitasking.
- However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- Threads are independent. If there occurs exception in one thread, it doesn't affect other threads.
- Java Multithreading is mostly used in games, animation, etc.

Advantages of Java Multithreading

1. It doesn't block the user because threads are independent, and you can perform multiple operations at the same time.
2. You can perform many operations together, so it saves time**.**
3. Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.
4. Threads share the same address space.
5. A thread is lightweight.
6. Cost of communication between the thread is low.

Creating a thread

There are two ways to create a thread:

- By extending Thread class
- By implementing Runnable interface.

Thread class

Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Commonly used methods of Thread class:

Modifier and Type	Method	Description
void	start()	It is used to start the execution of the thread.
void	run()	It is used to do an action for a thread.
static void	sleep()	It sleeps a thread for the specified amount of time.
static Thread	currentThread()	It returns a reference to the currently executing thread object.
void	join()	It waits for a thread to die.
int	getPriority()	It returns the priority of the thread.
void	setPriority()	It changes the priority of the thread.
String	getName()	It returns the name of the thread.
void	setName()	It changes the name of the thread.
long	getId()	It returns the id of the thread.
boolean	isAlive()	It tests if the thread is alive.
static void	yield()	It causes the currently executing thread object to pause and allow other threads to execute temporarily.
void	suspend()	It is used to suspend the thread.
void	resume()	It is used to resume the suspended thread.
void	stop()	It is used to stop the thread.
void	destroy()	It is used to destroy the thread group and all of its subgroups.
boolean	isDaemon()	It tests if the thread is a daemon thread.
void	setDaemon()	It marks the thread as daemon or user thread.
void	interrupt()	It interrupts the thread.
boolean	isinterrupted()	It tests whether the thread has been interrupted.
static boolean	interrupted()	It tests whether the current thread has been interrupted.

1) Java Thread Example by extending Thread class

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
}
}
```

Output

```
thread is running...
```

2) Java Thread Example by implementing Runnable interface

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

public static void main(String args[]){
Multi m1=new Multi();
Thread t1 =new Thread(m1);    //NEW state
t1.start();  //RUNNABLE state
}
}
```

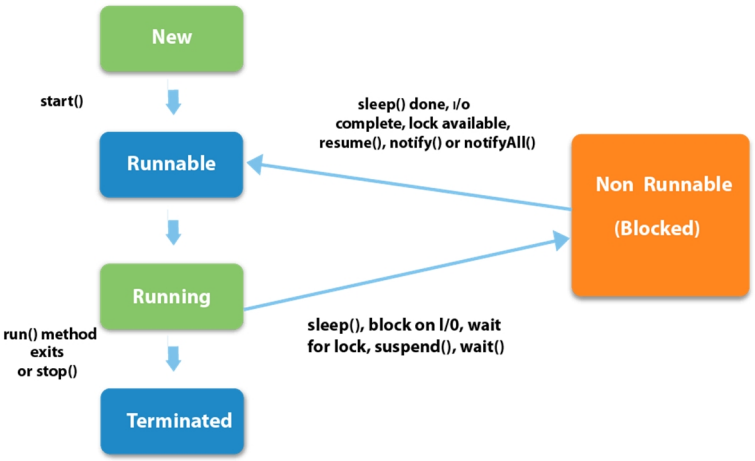
Output

thread is running...

Life cycle of a Thread

A thread can be in one of the five states. The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



1) New
The thread is in new state if you create an instance of Thread class but before the invocation of start() method.
2) Runnable
The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.
3) Running
The thread is in running state if the thread scheduler has selected it.
4) Non-Runnable (Blocked)
This is the state when the thread is still alive but is currently not eligible to run.
5) Terminated
A thread is in terminated or dead state when its run() method exits.

4.Explain thread priority with an example.

Priority of a Thread

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

- public static int MIN_PRIORITY
- public static int NORM_PRIORITY
- public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread

```
class TestMultiPriority extends Thread{
public void run(){
System.out.println("running thread name is:"+Thread.currentThread().getName());
System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
}
public static void main(String args[]){
TestMultiPriority m1=new TestMultiPriority1();
TestMultiPriority m2=new TestMultiPriority1();
m1.setPriority(Thread.MIN_PRIORITY);
m2.setPriority(Thread.MAX_PRIORITY);
m1.start();
m2.start();
}
}
```

O/P:
running thread name is:Thread-0
running thread priority is:1
running thread name is:Thread-1
running thread priority is:10

5.What are exceptions in Thread?

There are different exceptions in Thread:

IllegalThreadStateException comes in two situations,

1. Calling a start() on the thread when it's already executing run() method -:
   When a thread is already running and executing its functions inside the run() method and amidst its execution, a call to the start() method on the same thread, leads to an IllegalThreadStateException.

2. Calling start() on thread when it has finished executing run() method -:
   Once a thread finishes its execution, it cannot be started again, i.e. it is just dead. Hence, when a thread has already finished executing the run() method, calling the start() method on the same dead thread object(because someone feels it will restart a dead thread), leads to an IllegalTheadStateException.

Interrupting a Thread:

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true.

Eg:

```
class TestInterruptingThread extends Thread{
public void run(){
try{
Thread.sleep(1000);
System.out.println("task");
}catch(InterruptedException e){
System.out.println("Exception handled "+e);
}
System.out.println("thread is running...");
}
public static void main(String args[]){
TestInterruptingThread t1=new TestInterruptingThread();
t1.start();
t1.interrupt();
}
}
```

Output:

```
Exception handled
java.lang.InterruptedException: sleep interrupted
thread is running...
```

6.Explain Thread Synchronization with examples.

Synchronization in Java

Synchronization in java is the capability to control the access of multiple threads to any shared resource. Synchronization is a better option where we want to allow only one thread to access the shared resource.

The synchronization is mainly used to

- To prevent thread interference.
- To prevent consistency problem.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Thread Synchronization

There are two types of thread synchronization:

1. Mutual Exclusive
   a) Synchronized method.
   b) Synchronized block.
   c) static synchronization.
2. Cooperation (Inter-thread communication in java)

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

Synchronized method

- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.

Eg:

```
class Table{

 synchronized void printTable(int n){    //synchronized method

….

}
```

Synchronized block

- Synchronized block is used to lock an object for any shared resource.
- Synchronized block can be used to perform synchronization on any specific resource of the method.
- Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

Eg:

```
class Table{

 void printTable(int n){

   synchronized(this){    //synchronized block

….

}
```

Static Synchronization

- If you make any static method as synchronized, the lock will be on the class not on object.

Eg:

Without synchronization

```
class Table{
 static void printTable(int n){
   for(int i=1;i<=2;i++){
     System.out.println(n*i);
     try{
       Thread.sleep(400);
     }catch(Exception e){}
   }
 }
}
```

```
class MyThread1 extends Thread{
public void run(){
Table.printTable(1);
} }

class MyThread2 extends Thread{
public void run(){
Table.printTable(10);
} }

public class TestSynchronization4{
public static void main(String t[]){
MyThread1 t1=new MyThread1();
MyThread2 t2=new MyThread2();

t1.start();
t2.start();
}
}
```

O/P:
1
10
2
20

With Static synchronization

```
class Table{

 synchronized static void printTable(int n){
   for(int i=1;i<=2;i++){
     System.out.println(n*i);
     try{
       Thread.sleep(400);
     }catch(Exception e){}
   }
 }
}

class MyThread1 extends Thread{
public void run(){
Table.printTable(1);
} }

class MyThread2 extends Thread{
public void run(){
Table.printTable(10);
} }

public class TestSynchronization4{
public static void main(String t[]){
MyThread1 t1=new MyThread1();
MyThread2 t2=new MyThread2();

t1.start();
t2.start();
}
}
```

O/P:
1
2
10
20

Inter-thread communication in Java

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.
Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of Object class:

- wait()
- notify()
- notifyAll()

MODULE IV

1.What are streams? Explain stream related classes in Java. Write an example for file I/O in Java using streams.

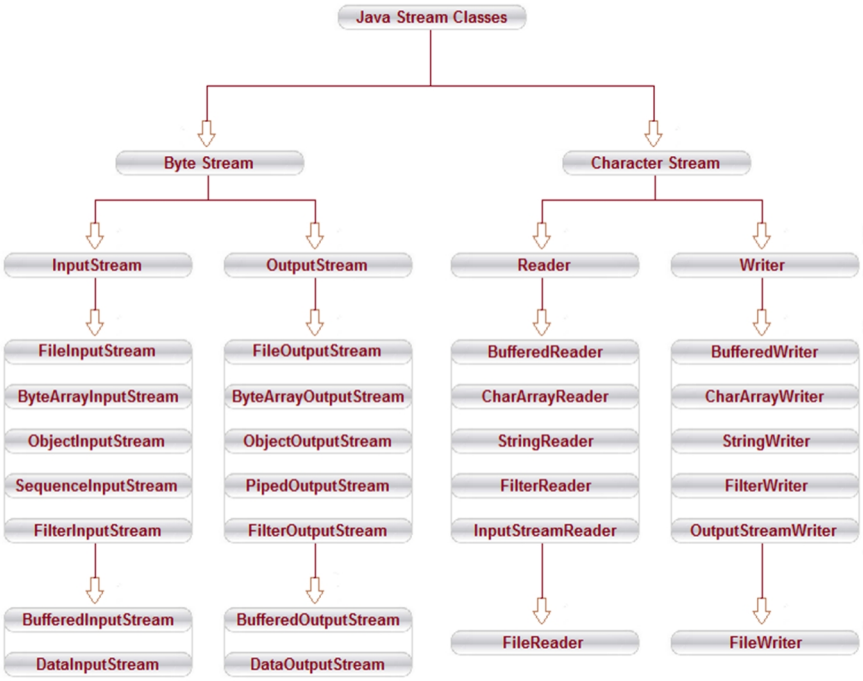
Stream Classes

In Java, a **stream** is a **path** along which the data flows. Every stream has a source and a destination. We can build a complex file processing sequence using a series of simple stream operations. Two fundamental types of streams are Writing streams and Reading streams. While writing streams writes data into a file, reading streams reads data from a file.

Types of Streams

The java.io package contains a large number of stream classes that provide capabilities for processing all types of data. These classes may be categorized into two groups based on the data type on which they operate.

- Byte stream classes
- Character stream classes



Byte Stream Classes

Byte stream classes have been designed to provide functional features for creating and manipulating streams and files for reading and writing bytes.
Java Byte streams are used to perform input and output of 8-bit bytes. Java provides two kinds of byte stream classes:

1. **Input Stream Classes**

Input stream classes that are used to read bytes include a super class known as Inputstream and a number of subclasses for supporting various input-related functions. The super class InputStream is an abstract class, and, therefore, we cannot create instances of this class. Rather, we must use the subclasses that inherit from this class.

2. **Output Stream Classes**

Output stream classes are derived from the base class Outputstream. Like InputStream, the OutputStream is an abstract class and therefore we cannot instantiate it. The several subclasses of the OutputStream can be used for performing the output operations.

Below are the subclasses of Input stream and Output stream classes:

- **FilterInputStream:** An instance of this class contains some other input stream as a basic source of data for further manipulation.
- **FileOutputStream:** An instance of this class is used to output a stream for writing data to a file or to a file descriptor.

- **ByteArrayInputStream:** An instance of this class contains an internal buffer to read bytes stream.
- **ByteArrayOutputStream:** An instance of this class contains an internal buffer to write a bytes stream.

- **ObjectInputStream:** An instance of this class is used to deserialize an object after it has been serialized by ObjectOutputSteam.
- **ObjectOutputStream:** An instance of this class is used to serialize an object which can be deserialized with *ObjectInputStream*.

- **SequenceInputStream:** An instance of this class represents a logical concatenation of two or more input streams which are read in sequence, one after another.

- **PipedInputStream:** An instance of this class provides a pipe or buffer for an input byte that works in the FIFO manner.
- **PipedOutputStream:** An instance of this class provides a pipe or buffer for output byte that works in the FIFO manner.

- **FilterInputStream:** An instance of this class contains some other input stream as a basic source of data for further manipulation.
  - **BufferedInputStream:** This enables a *FilterInputStream* instance to make use of a buffer for input data.
  - **DataInputStream:** An instance of this class enables reading primitive Java types from an underlying input stream in a machine-independent manner.
- **FilterOutputStream:** An instance of this class contains some other output stream as a basic source of data for further manipulation.
  - **BufferedOutputStream:** This enables a *FilterOutputStream* instance to make use of a buffer for output data.
  - **DataOutputStream:** An instance of this class enables writing primitive Java types to an underlying output stream in a machine-independent manner.

Character Stream Classes

Character streams can be used to read and write 16-bit Unicode characters. Java Character streams are used to perform input and output for 16-bit Unicode. Like byte streams, there are two kinds of character stream classes:

1. **Reader Stream Classes**

Reader stream classes that are used to read characters include a super class known as Reader and a number of subclasses for supporting various input-related functions. Reader stream classes are functionally very similar to the input stream classes, except input streams use bytes as their fundamental unit of information, while reader streams use characters.

2. **Writer Stream Classes**

Like output stream classes, the writer stream classes are designed to perform all output operations on files. Only difference is that while output stream classes are designed to write bytes, the writer stream are designed to write character.

Below are the subclasses of Reader stream and Writer stream classes:

- **BufferedReader:** Provides an in-between buffer for efficiency while reading text from character input stream.
- **BufferedWriter:** Provides an in-between buffer for efficiency while writing text to a character output stream.

- **CharArrayReader:** Implements an auto-increasing character buffer that may be used as a reader.
- **CharArrayWriter:** Implements an auto-increasing character buffer that may be used as a writer.

- **StringReader:** Character output stream reader from source string.
- **StringWriter:** Character output stream is collected in a string buffer and may be used for constructing a string.

- **FilterReader:** An instance of this class is used for reading character files.
- **FilterWriter:** Abstract class for writing filtered character streams.

- **InputStreamReader:** An instance of this class provides a bridge from byte streams to character streams. Bytes are decoded into characters using a specified character set.
  - **FileReader:** An instance of this class is used for reading character files.

- **OutputStreamWriter:** An instance of this class provides a bridge between character streams and byte streams. Characters are encoded into bytes using a specified character set.
  - **FileWriter:** An instance of this class is used for writing character files.

Example for file I/O in Java using stream

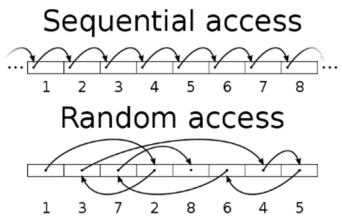
```
import java.io.*;
public class CopyFile {
public static void main(String args[]) throws IOException {
FileInputStream in = null;
FileOutputStream out = null;
try {
in = new FileInputStream("input.txt");
out = new FileOutputStream("output.txt");
int c;
while ((c = in.read()) != -1) {
out.write(c);
}
}
finally{
if (in != null) {
in.close();
}
if (out != null) {
out.close();
} } }
```

Now let's have a file input.txt with the following content −
This is test for copy file.
After compiling the above program and execute it, will result in creating output.txt file with the same content as we have in input.txt.

2.What are Random Access Files in Java?

Random Access File

Java **Random Access File** provides the facility to read and write data to a file. Random Access File works with file as large array of bytes stored in the file system and a cursor using which we can move the file pointer position.



Following is the declaration for Java.io.RandomAccessFile class −

public class RandomAccessFile extends Object implements DataOutput, DataInput, Closeable

A Random Access File can be created in four different access modes. The access mode value is a string. They are listed as follows:

"r"	The file is opened in a read-only mode.
"rw"	The file is opened in a read-write mode. The file is created if it does not exist.
"rws"	The file is opened in a read-write mode. Any modifications to the file's content and its metadata are written to the storage device immediately.
"rwd"	The file is opened in a read-write mode. Any modifications to the file's content are written to the storage device immediately.

Eg:

```
RandomAccessFile  raf = new RandomAccessFile("randomtext.txt", "rw");
```

3.What are the features of AWT? Explain Container class and Frame window. How frame is created and information placed in it? Illustrate with the help of a simple AWT program.

AWT

Java AWT (Abstract Window Toolkit) is an API to develop GUI or window-based applications in java.

Java AWT components are platform dependent i.e. components are displayed according to the view of operating system.

AWT is heavyweight i.e. its components are using the resources of OS.

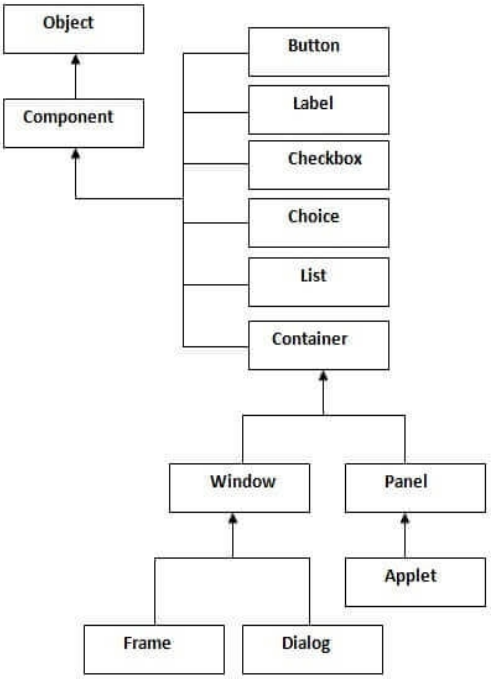
The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

AWT features include:

- A set of native user interface components
- A robust event-handling model
- Graphics and imaging tools, including shape, color, and font classes
- Layout managers, for flexible window layouts that do not depend on a particular window size or screen resolution
- Data transfer classes, for cut-and-paste through the native platform clipboard

Java AWT Hierarchy

The hierarchy of Java AWT classes are given below.



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.

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.

Panel

The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc.

Frame

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.
Creating a Frame
There are two ways to create a Frame. They are,

1. By Instantiating Frame class
2. By extending Frame class

Creating Frame Window by Instantiating Frame class

```
import java.awt.*;
public class Testawt
{
 Testawt()
 {
   Frame f=new Frame();    //Creating a frame
   Label l1 = new Label("welcome to java");   //Creating a label
   f.add(l1);      //adding label to the frame
   f.setSize(300, 300);   //setting frame size.
   f.setVisible(true);     //set frame visibility true
 }
 public static void main(String args[])
 {
   Testawt ta = new Testawt();
 }
}
```

Creating Frame window by extending Frame class

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

public class Testawt extends Frame
{
  public Testawt()
  {
    Button btn=new Button("Click here");
    add(btn);  //adding a new Button.
    setSize(400, 500);      //setting size.
    setTitle("New Frame");  //setting title.
    setLayout(new FlowLayout());    //set default layout for frame.
    setVisible(true);      //set frame visibilty true.
  }

  public static void main (String[] args)
  {
    Testawt ta = new Testawt();  //creating a frame.
  }
}
```

4.Explain event handling in AWT programming with an example (Delegation Event Model).

Event

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc

Event Handling

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has the code which is known as event handler that is executed when an event occurs. Java Uses the **Delegation Event Model** to handle the events. This model defines the standard mechanism to generate and handle the events.

The Delegation Event Model has the following key participants namely:
Source - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler.
Listener - It is also known as event handler. Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received, the listener processes the event and then returns.

Java Event classes and Listener interfaces

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

Steps involved in event handling

1. The User clicks the button and the event is generated.
2. Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
3. Event object is forwarded to the method of registered listener class.
4. The method now gets executed and returns.

Java event handling by implementing ActionListener

```
import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame implements ActionListener{
TextField tf;
AEvent(){

//create components
tf=new TextField();
tf.setBounds(60,50,170,20);
```

```
Button b=new Button("click me");
b.setBounds(100,120,80,30);

//register listener
b.addActionListener(this);//passing current instance

//add components and set size, layout and visibility
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
}
public static void main(String args[]){
new AEvent();
}
}
```

5.What is Applet Class? Explain Applet life cycle. What is the use of Graphics class in Java? How parameters are passed to Applet?

Java Applet

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

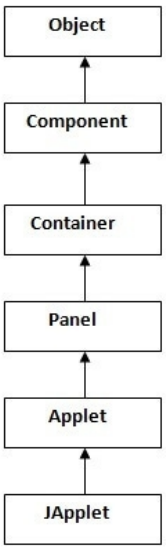
Advantage of Applet

- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many platforms, including Linux, Windows, Mac Os etc.

Drawback of Applet

- Plugin is required at client browser to execute applet. It manages the lifecycle of an Applet.

Hierarchy of Applet



As displayed in the above diagram, Applet class extends Panel. Panel class extends Container which is the subclass of Component.

Lifecycle of Java Applet

1. **Applet is initialized.**
2. **Applet is started.**
3. **Applet is painted.**
4. **Applet is stopped.**
5. **Applet is destroyed.**

Lifecycle methods for Applet

The java.applet.Applet class have four life cycle methods and java.awt.Component class provides one life cycle methods for an applet.

java.applet.Applet class

- **public void init():** is used to initialized the Applet. It is invoked only once.
- **public void start():** is invoked after the init() method or browser is maximized. It is used to start the Applet.
- **public void stop():** is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
- **public void destroy():** is used to destroy the Applet. It is invoked only once.

java.awt.Component class

- **public void paint(Graphics g):** is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.

There are two ways to run an applet
1. By html file.
2. By appletViewer tool (for testing purpose).

Simple example of Applet by html file

```
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{
public void paint(Graphics g){
g.drawString("welcome",150,150);
}
}
```

myapplet.html

```
<html>
<body>
<applet code="First.class" width="300" height="300">
</applet>
</body>
</html>
```

Displaying Graphics in Applet

java.awt.Graphics class provides many methods for graphics programming.

Commonly used methods of Graphics class:
public void drawString(String str, int x, int y): is used to draw the specified string.
public void drawOval(int x, int y, int width, int height): is used to draw oval with the specified width and height.
public void fillOval(int x, int y, int width, int height): is used to fill oval with the default color and specified width and height.
public void drawLine(int x1, int y1, int x2, int y2): is used to draw line between the points(x1, y1) and (x2, y2).
public void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle): is used draw a circular or elliptical arc.
public void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle): is used to fill a circular or elliptical arc.
public void setColor(Color c): is used to set the graphics current color to the specified color.
public void setFont(Font font): is used to set the graphics current font to the specified font.

Example of Graphics in applet

```
import java.applet.Applet;
import java.awt.*;

public class GraphicsDemo extends Applet{
public void paint(Graphics g){
g.setColor(Color.red);
g.drawString("Welcome",50, 50);
g.drawOval(70,200,30,30);
g.setColor(Color.pink);
g.fillOval(170,200,30,30);
} }
```

myapplet.html

```
<html>
<body>
<applet code="GraphicsDemo.class" width="300" height="300">
</applet>
</body>
</html>
```

Parameter in Applet

We can get any information from the HTML file as a parameter. For this purpose, Applet class provides a method named getParameter().

Syntax:
public String getParameter(String parameterName)

Example of using parameter in Applet

```
import java.applet.Applet;
import java.awt.Graphics;

public class UseParam extends Applet{
public void paint(Graphics g){
String str=getParameter("msg");
g.drawString(str,50, 50);
}
}
```

myapplet.html

```
<html>
<body>
<applet code="UseParam.class" width="300" height="300">
<param name="msg" value="Welcome to applet">
</applet>
</body>
</html>
```

MODULE V

1.Explain JDBC architecture.

JDBC

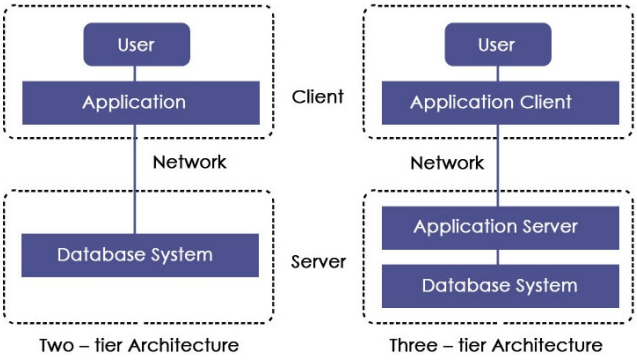
JDBC stands for **Java Database Connectivity**. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

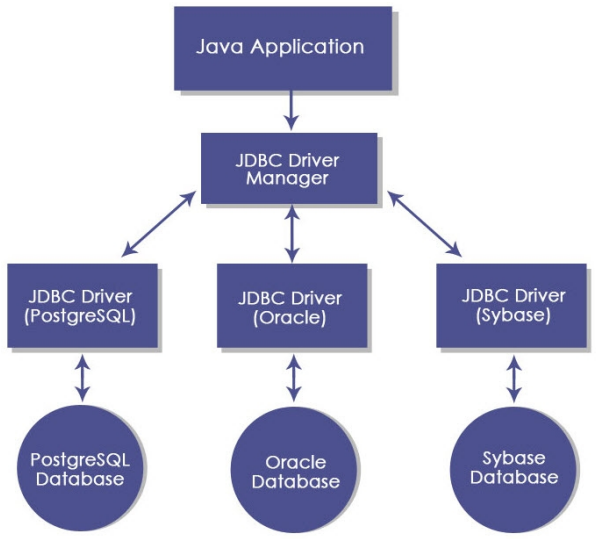
- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database.

JDBC Architecture



The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers –
JDBC API: This provides the application-to-JDBC Manager connection. It uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.
JDBC Driver API: This supports the JDBC Manager-to-Driver Connection. It ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.



Common JDBC Components

The JDBC API provides the following interfaces and classes –

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.

- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

JDBC Drivers

There are four types of JDBC drivers:

- **JDBC-ODBC Bridge Driver** - This driver acts as a bridge between JDBC and ODBC. It converts JDBC calls into ODBC calls and then sends the request to ODBC driver. They are very slow.
- **Native Driver** - This driver uses JNI (Java Native Interface) call on database specific native client API. They are comparatively faster than JDBC-ODBC Bridge Driver.
- **Network Protocol Driver** - These drivers communicate to JDBC middleware server using proprietary network protocol. This middleware translates the network protocol to database specific calls. They are database independent. They can switch from one database to another but are slow due to many network calls.
- **Thin Driver** - This driver is also called pure Java driver because they directly interact with the database. It neither requires any native library nor middleware server. They are fastest among those listed above.

2.Explain type of statements in JDBC.

There are three types of statements in JDBC namely,

- Statement
- Prepared Statement
- Callable statement

Statement

The Statement interface represents the static SQL statement. It helps you to create a general purpose SQL statements using Java.

The **createStatement()** method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

Example :
Statement stmt=con.createStatement();

Executing the Statement object

Once you have created the statement object you can execute it using one of the following methods.

- **execute():** This method is used to execute SQL DDL statements, it returns a boolean value specifying whether the ResultSet object can be retrieved.
- **executeUpdate():** This method is used to execute statements such as insert, update, delete. It returns an integer value representing the number of rows affected.
- **executeQuery():** This method is used to execute statements that returns tabular data (example SELECT statement). It returns an object of the class ResultSet.

Prepared Statement

The PreparedStatement interface extends the Statement interface. It represents a precompiled SQL statement which can be executed multiple times. This accepts parameterized SQL quires and you can pass 0 or more parameters to this query.

You can create an object of the **PreparedStatement** (interface) using the **prepareStatement()** method of the Connection interface. This method accepts a query (parameterized) and returns a PreparedStatement object.

Example :
PreparedStatement pstmt = con.prepareStatement(query);

Executing the Prepared Statement

Once you have created the PreparedStatement object you can execute it using one of the following methods.

- **execute():** This method executes normal static SQL statements in the current prepared statement object and returns a boolean value.
- **executeQuery():** This method executes the current prepared statement and returns a **ResultSet** object.
- **executeUpdate():** This method executes SQL DML statements such as insert update or delete in the current Prepared statement. It returns an integer value representing the number of rows affected.

Callable Statement

The CallableStatement interface provides methods to execute stored procedures.

You can create an object of the **CallableStatement** (interface) using the **prepareCall()** method of the **Connection** interface. This method accepts a string variable representing a query to call the stored procedure and returns a CallableStatement object.

Example :
CallableStatement cstmt = con.prepareCall("{call myProcedure(?, ?, ?)}");

Executing the Callable Statement

Once you have created the CallableStatement object you can execute it using **execute()** method.

3.What are the steps in connecting to a database? Illustrate with example.

Steps in connecting to a Database

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

1. Register the Driver class
2. Create connection
3. Create statement
4. Execute queries
5. Close connection

1) Register the driver class

The **forName()** method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Example :
Here, Java program is loading mysql driver to establish database connection.
Class.forName("com.mysql.jdbc.Driver");

2) Create the connection object

The **getConnection()** method of DriverManager class is used to establish connection with the database.

Example :
Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/student","root","password") // “url”,”username”,”password”

3) Create the Statement object

The **createStatement()** method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

Example :
Statement stmt=con.createStatement();

4) Execute the query

The **executeQuery()** method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

Example :
ResultSet rs=stmt.executeQuery("select * from studdetails");
while(rs.next()){
System.out.println(rs.getInt(1)+" "+rs.getString(2));
}

5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The **close()** method of Connection interface is used to close the connection.

Example :
con.close();

Java program for connecting to a MySQL database

```
public class studinfo {
public static void main(String[] args) {
try{
Class.forName("com.mysql.jdbc.Driver");
```

```
Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/student","root","password");
Statement stmt=con.createStatement();
String sql="INSERT INTO studdet VALUES ('Sarah',100)";
stmt.executeUpdate(sql);
System.out.println("inserted records into the table...");
ResultSet rs=stmt.executeQuery("SELECT * FROM studdet");
while(rs.next())
{
String name=rs.getString(1);
int rollno=rs.getInt(2);
System.out.println("Name:"+name);
System.out.println("Roll no:"+rollno);
}
}
catch(Exception e)
{
System.out.println("Exception"+e);
}
}
}
```

O/P:
inserted records into the table...
Name:Sarah
Roll no:100

IN Database,

*Select * from studdet;*

name	rollno
Sarah	100

4.What is Socket programming in Java? Write a program for Client-Server communication using socket.

Java Socket Programming

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request. **Java Socket programming** is used for communication between the applications running on different JRE.

Java Socket programming is of two types:

- **Connection oriented** - Socket and ServerSocket classes are used for connection-oriented socket programming.
- **Connection-less** - DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know two information:

1. IP Address of Server, and
2. Port number.

In one-way client and server communication, client sends a message to the server, server reads the message and prints it. Here, two classes are being used: Socket and ServerSocket. The **Socket** class is used to communicate client and server. Through this class, we can read and write message. The **ServerSocket** class is used at server-side. The accept() method of ServerSocket class blocks the console until the client is connected. After the successful connection of client, it returns the instance of Socket at server-side.

Socket class

A socket is simply an endpoint for communications between the machines. The Socket class can be used to create a socket.

Important methods in Socket class include:

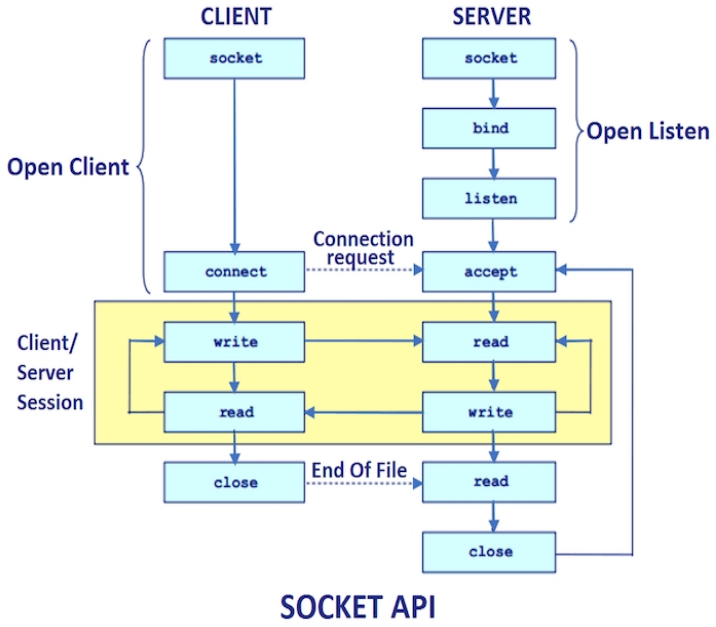
• public InputStream getInputStream()	returns the InputStream attached with this socket.
• public OutputStream getOutputStream()	returns the OutputStream attached with this socket.
• public synchronized void close()	closes this socket

ServerSocket class

The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients.

Important methods in ServerSocket class include:

• public Socket accept()	returns the socket and establish a connection between server and client.
• public synchronized void close()	closes the server socket.



Creating Server and Client application

Creating Server:

To create the server application, we need to create the instance of ServerSocket class. Here, we are using 8080 port number for the communication between the client and server. You may also choose any other port number. The accept() method waits for the client. If clients connects with the given port number, it returns an instance of Socket.

Eg:
ServerSocket ss=**new** ServerSocket(8080);
Socket s=ss.accept(); //establishes connection and waits for the client

Creating Client:

To create the client application, we need to create the instance of Socket class. Here, we need to pass the IP address or hostname of the Server and a port number. Here, we are using "localhost" because our server is running on same system.

Socket s=**new** Socket("localhost",8080);

Example of socket programming where client sends a text and server receives and prints it.

Server application

```
import java.io.*;
import java.net.*;
public class MyServer {
public static void main(String[] args){
try{
ServerSocket ss=new ServerSocket(8080);
Socket s=ss.accept();      //establishes connection
DataInputStream dis=new DataInputStream(s.getInputStream());
String  str=(String)dis.readUTF();
System.out.println("message= "+str);
ss.close();      //connection closed
}
catch(Exception e){
System.out.println(e);
} }
}
```

Client application

```
import java.io.*;
import java.net.*;
public class MyClient {
public static void main(String[] args) {
try{
Socket s=new Socket("localhost",8080);
DataOutputStream dos=new DataOutputStream(s.getOutputStream());
```

```
dos.writeUTF("Hello Server");
dos.flush();
dos.close();
s.close();
}
catch(Exception e){
System.out.println(e);
} }
}
```

5.Explain the following terms: i) InetAddress ii) URL Connection iii) Datagram.

i) InetAddress

Java **InetAddress** class represents an IP address. The java.net.InetAddress class provides methods to get the IP of any host name *for example* www.google.com. An IP address is represented by 32-bit or 128-bit unsigned number. An instance of InetAddress represents the IP address with its corresponding host name.

Moreover, InetAddress has a cache mechanism to store successful and unsuccessful host name resolutions.

Commonly used methods of InetAddress class are:

public static InetAddress getByName(String host)	- returns the instance of InetAddress containing localhost IP and name.
public static InetAddress getLocalHost()	- returns the instance of InetAddress containing localhost name and address.
public String getHostName()	- returns the host name of the IP address.
public String getHostAddress()	- returns the IP address in string format.

Example:

```
import java.io.*;
import java.net.*;
public class InetDemo{
public static void main(String[] args){
try{
InetAddress ip=InetAddress.getByName("www.google.com");
System.out.println("Host Name: "+ip.getHostName());
System.out.println("IP Address: "+ip.getHostAddress());
}
catch(Exception e){
System.out.println(e);
} }
}
```

Output:
Host Name: www.google.com
IP Address: 201.23.11.66

ii) Java URL

The **Java URL** class represents a URL. URL is an acronym for Uniform Resource Locator. It points to a resource on the World Wide Web.
For example: https://www.google.com:80/images

A URL contains many information:

Protocol: In this case, http is the protocol.
Server name or IP Address: In this case, www.google.com is the server name.
Port Number: It is an optional attribute. In this case, 80 is the port number. If port number is not mentioned in the URL, it returns -1.
File Name or directory name: In this case, images is the file name.

Commonly used methods of Java URL class are:

public String getProtocol()	- it returns the protocol of the URL.
public String getHost()	- it returns the host name of the URL.
public String getPort()	- it returns the Port Number of the URL.
public String getFile()	- it returns the file name of the URL.

Example:

```
import java.net.*;
public class URLDemo{
public static void main(String[] args){
try{
URL url=new URL("http://www.google.com:80/images");
System.out.println("Protocol: "+url.getProtocol());
System.out.println("Host Name: "+url.getHost());
```

```
System.out.println("Port Number: "+url.getPort());
System.out.println("File Name: "+url.getFile());
}
catch(Exception e){
System.out.println(e);
} }
}
```

Output:
Protocol: http
Host Name: www.google.com
Port Number: 80
File Name: /images

iii) Datagram

A **datagram** is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.

DatagramSocket and **DatagramPacket**

Java DatagramSocket and DatagramPacket classes are used for connection-less socket programming.
Java DatagramSocket class represents a connection-less socket for sending and receiving datagram packets.
Java DatagramPacket class represents a message that can be sent or received. If you send multiple packet, it may arrive in any order. Additionally, packet delivery is not guaranteed.

Example of Sending DatagramPacket by DatagramSocket

```
import java.net.*;
public class DSender{
  public static void main(String[] args) throws Exception {
    DatagramSocket ds = new DatagramSocket();
    String str = "Welcome java";
    InetAddress ip = InetAddress.getByName("127.0.0.1");
    DatagramPacket dp = new DatagramPacket(str.getBytes(), str.length(), ip, 3000);
    ds.send(dp);
    ds.close();
  }
}
```

Example of Receiving DatagramPacket by DatagramSocket

```
import java.net.*;
public class DReceiver{
  public static void main(String[] args) throws Exception {
    DatagramSocket ds = new DatagramSocket(3000);
    byte[] buf = new byte[1024];
    DatagramPacket dp = new DatagramPacket(buf, 1024);
    ds.receive(dp);
    String str = new String(dp.getData(), 0, dp.getLength());
    System.out.println(str);
    ds.close();
  }
}
```