TOPS TECHNOLOGY

Training | Outsourcing | Placement | Study Abroad

Java

Assignment :1

1.History of Java

- →**1991**: Java started as a project called **Oak** by **James Gosling** at **Sun Microsystems**, meant for embedded systems like TVs.
- **1995**: Oak was renamed to **Java** and officially launched. Its main feature was platform independence

Late 1990s: Java became popular for web applications enterprise software, and mobile phones (J2ME).

- **2006**: Java became **open-source** through the **OpenJDK** project.
- **2010**: **Oracle Corporation** acquired **Sun Microsystems** and took over Java's development.
- **2014**: **Java 8** introduced big features like **Lambdas** and **Streams**.
- **2017–Now**: Java follows a **6-month release cycle**, with regular updates and modern features .

2.Features of Java (Platform Independent, Object-Oriented, etc.)

1. →**Platform Independent**
   - Write code once, run it anywhere using the Java Virtual Machine (JVM).
2. **Object-Oriented**
   - Everything in Java is treated as an object; supports concepts like class, inheritance, polymorphism.
3. **Simple**
   - Easy to learn and use; syntax is clean and similar to C++ but without complex features like pointers.
4. **Secure**

    - Has built-in security features; runs code inside a secure JVM environment.

5. **Robust**
    - Strong memory management, exception handling, and no direct memory access (no pointers).
6. **Portable**
    - Java programs can run on any device with a JVM, without changes.
7. **High Performance**
    - Faster than other interpreted languages due to Just-In-Time (JIT) compiler.
8. **Multithreaded**
    - Supports multithreading (running multiple tasks at once) for better performance.
9. **Distributed**
    - Can build distributed applications using tools like RMI and EJB.

10.Dynamic

- Java loads classes at runtime, making it flexible and extensible.

3. Understanding JVM, JRE, and JDK

→1)**JVM (Java Virtual Machine)**

- Runs Java programs.
- Converts bytecode to machine code.
- Makes Java platform independent.

2) JRE (Java Runtime Environment)

- Contains **JVM + libraries**.
- Used to **run** Java programs.

- Cannot write or compile code.

3)JDK (Java Development Kit)

- Contains **JRE + tools** (like compiler).
- Used to **write, compile, and run** Java programs.
- Needed by Java developers.

4

→1. Install JDK

- Download from: oracle.com/java
- Install it on your computer.
- To check:
  Open terminal or command prompt and type:
  java -version
  javac -version

2. Install IDE

Eclipse

- Download from: eclipse.org
- Install and open.
- Go to: File → New → Java Project

IntelliJ IDEA

- Download from: jetbrains.com/idea
- Choose **Community Edition** (free).
- Open and go to: File → New Project → Java.

5. Java Program Structure (Packages, Classes, Methods)

→

```
package mypackage;


public class MyClass {
    public static void main(String[] args) {
        System.out.println("Hello, Java!");
    }


    void myMethod() {     // Another method
        // Code here
    }
}
```

1. Package

- Used to group related classes.
- Example: package mypackage;

2. Class

- The main building block.
- Every program has at least one class.
- Example: public class MyClass { }

3. Method

- Block of code that performs actions.
- main() is the entry point.

Ex:

```
public static void main(String[] args) {
   // program starts here
}
```

6) Primitive Data Types in Java (int, float, char, etc.)

→

Data Type	**Size**	**Example**	**Use**
byte	1 byte	byte b = 10;	Very small integers (-128 to 127)
short	2 bytes	short s = 200;	Small integers
int	4 bytes	int i = 1000;	Default whole numbers
long	8 bytes	long l = 10000L;	Large whole numbers
float	4 bytes	float f = 5.5f;	Decimal numbers (less precise)
double	8 bytes	double d = 9.99;	Decimal numbers (more precise)
char	2 bytes	char c = 'A';	Single character
boolean	1 bit	boolean flag = true;	True or False

7) Variable Declaration and Initialization.

→ means is nothing but to store some value.

Declaration: Telling the compiler about the variable type and name.
Example:

Ex:

```
int age;
```

Initialization: Assigning a value to the variable.

Ex:

```
   age = 25;
```

Declaration and Initialization together:

Ex:

```
   int age = 25;
```

8) Operators: Arithmetic, Relational, Logical, Assignment, Unary, and Bitwise .

→

Operator Type	**Description**	**Examples**
Arithmetic	Perform math operations	+, -, *, /, %
Relational	Compare values	==, !=, <, >, <=, >=
Bitwise	Operate on bits	& (AND), `
Logical	Combine boolean values	&& (AND), `
Assignment	Assign values	=, +=, -=, *=, /=
Unary	Single operand operations	++ (increment), -- (decrement), +, -, !

Examples:

1)Arithmetic:

→int sum = 5 + 3; // 8

2) Relational:

→boolean result = (5 > 3);

3) Logical:

→boolean flag = (5 > 3) && (3 < 7);

4) Assignment:

→ int a = 10;

a += 5;

5) Unary:

→ int x = 5;

x++;

6) Bitwise:

→int b = 5 & 3;

9) Type Conversion and Type Casting

→convert from one data type to another data type mechanism

1. Type Conversion (Implicit)

- Automatic conversion by Java when assigning smaller type to a bigger type.
- Also called **widening conversion**.
- No data loss.

Ex:

```
int i = 100;
 long l = i;
```

2. Type Casting (Explicit)

- Manually converting one data type to another.
- Needed when converting bigger type to smaller type.
- Can cause data loss.

Ex:

```
double d = 9.78;
 int i = (int) d;
```

10) If-Else Statements.

→if condition is true then your if block will be execute otherwise else block will be executed.

Syntax:

```
 if (condition) {
   // code runs if condition is true
} else {
   // code runs if condition is false
}
```

EX:

```
int age = 18;


if (age >= 18) {
   System.out.println("You are an adult.");
} else {
   System.out.println("You are a minor.");
}
```

11) Switch Case Statements

→executes one statement from multiple ones. Thus, it is like an if-else-if ladder statement.

Syntax:

```
 switch (variable) {
    case value1:
        // code block
        break;
    case value2:
        // code block
        break;
    default:
        // code if no case matches
}
```

Ex:

```
    int day = 3;

switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    default:
        System.out.println("Invalid day");
```

}

12) Loops (For, While, Do-While)

→1) entry control loop : 1.while , 2.for

2) exit control loop : 1.dowhile

1)**For Loop**

→ Used when you know how many times to repeat.

Syntax:

```
for (initialization; condition; update) {
// code to repeat
}
```

Ex:

```
for (int i = 1; i <= 5; i++) {
 System.out.println(i);
}
```

Output: 1 2 3 4 5

2. While Loop

→ Used when the number of repetitions is unknown.

Syntax:

```
while (condition) {
        // code to repeat
       }
```

Ex:

```
int i = 1;
```

```
while (i <= 5) {
   System.out.println(i);
   i++;
}
```

Output: 1 2 3 4 5

3. Do-While Loop

→ Same as while loop, but runs at least once.

Syntax:

```
     do {
   // code to repeat
} while (condition);
```

Ex:

```
int i = 1;
do {
   System.out.println(i);
   i++;
} while (i <= 5);
```

Output: 1 2 3 4 5

13) Break and Continue Keywords

→

1) break Keyword:

Ex:

```
     for (int i = 1; i <= 5; i++) {
   if (i == 3) {
      break;  // loop ends when i is 3
   }
   System.out.println(i);
}
```

Output: 1 2

2) continue Keyword:

Ex:

```
      for (int i = 1; i <= 5; i++)
    {
      if (i == 3)
     {
      continue;  // skip when i is 3
     }
     System.out.println(i);
   }
```

Output: 1 2 4 5

14) Defining a Class and Object in Java

→ Class is an collection of data member(variables) and member function(methods, process) with its behaviors.

sy:

```
class classname
{
          data member
          member function
          }
```

Ex:

```
          class Car {
        String color = "Red";


      void drive() {
       System.out.println("Car is driving");
     }
     }
```

2. Object

→ An object is an instance of a class.

Sy:

```
        classname objectname = new constructor();
```

Example:

```
        public class Main {
   public static void main(String[] args) {
       Car myCar = new Car();  // creating object
       System.out.println(myCar.color);  // access variable
       myCar.drive();  // call method
   }
}
```

15) Constructors and Overloading.

→ **1. Constructor:**A **constructor** is a special method that runs when an object is created.

Key Points:

- Same name as the class.
- No return type .
- Used to initialize objects.

Ex:

```
class Car {
   Car() {  // Constructor
      System.out.println("Car is created");
   }
}

public class Main {
   public static void main(String[] args) {
      Car c = new Car();  // Constructor is called
   }
}
```

2. Constructor Overloading

→ Using **multiple constructors** with different parameters in the same class.

Ex:

```
class Car {
   Car() {
      System.out.println("Default Car");
   }

   Car(String model) {
```

```
            System.out.println("Car Model: " + model);
        }
    }

    public class Main {
        public static void main(String[] args) {
            Car c1 = new Car();           // Calls default constructor
            Car c2 = new Car("Toyota");    // Calls parameterized constructor
        }
    }
```

16) Object Creation, Accessing Members of the Class.

→ You create an object using the new keyword.

Sy:

```
    ClassName obj = new ClassName();
```

Example:

```
      Car myCar = new Car();
```

2. Accessing Class Members

→ Use the **dot (.) operator** to access variables and methods of the class.

Ex:

```
    class Car {
   String color = "Red";         // variable
   void drive() {               // method
      System.out.println("Driving...");
   }
}
```

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();          // object created
        System.out.println(myCar.color); // access variable
        myCar.drive();                  // call method
    }
}
```

17) this Keyword

→ when your class variable name and argument variable names are same at that time to seprate your class variable with using this keyword : current class refernece

this means **"this object"** – the current object.

It is used **inside a class** to refer to **its own variables or methods**.

Use

1.When variable names are same:

```
     class Student {
    String name;

    Student(String name) {
        this.name = name;  // this refers to the object's variable
    }
}
```

2. To call another constructor:

```
class Student {
    Student() {
        this("John");  // calls another constructor
    }

    Student(String name) {
        System.out.println(name);
    }
}
```

18) Defining Methods.

→ A **method** is a block of code that performs a specific task.

Sy:

```
 returnType methodName(parameters) {
    // code to run
   }
```

Ex:

```
class MyClass {
    void sayHello() {
        System.out.println("Hello!");
    }

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

Using the Methods:

```
 public class Main {
   public static void main(String[] args) {
     MyClass obj = new MyClass();   // create object
     obj.sayHello();              // call method
     int sum = obj.add(5, 3);      // call method with return
     System.out.println(sum);
   }
}
```

19) Method Parameters and Return Types

→ Values passed **into** a method inside ().

Return Types

- Type of value a method **returns**.
- Use void if no value is returned.

Ex:

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


void greet(String name) {
   System.out.println("Hello, " + name);
}
```

20) Method Overloading

→ **Multiple methods** with the **same name** but **different parameters** (different type or number).

Helps perform similar actions in different ways.

Ex:

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

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

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

21)Static Methods and Variables

→1. **Static Variables**

- Belong to the **class**, not to objects.
- Shared by all objects of the class

2.Static Methods

- Belong to the **class**, not to objects.
- Can be called without creating an object.

Ex:

```
class MyClass {
    static int count = 0;


    static void display()
 {
        System.out.println("Count: " + count);
    }
}


public class Main {
    public static void main(String[] args)
{
        MyClass.count = 5;
        MyClass.display();
    }
}
```

22) Basics of OOP: Encapsulation, Inheritance, Polymorphism, Abstraction

→ 1. **Encapsulation**

- wrapping up of data into single unit
- data hiding
- private your data member and meber function

Use **private** variables and **public** getters/setters to protect data.

2. **Inheritance** :

- **properties of superclass extends into subclass**
- **main purpose is : Reusablity , extendsiblity**
- **to used "extends" keyword through create inheritance**

- **always called last child class to create object with access the properties of parent class except private**
- properties of parent class extends into child class

3. **Polymorphism**

- **bility to take one name having many forms or different forms**

Two types:

- **Method Overloading** (same class, same method name, different parameters)
- **Method Overriding** (child class changes parent's method)

4. Abstraction

- Hiding complex details and showing only important features

1) using with class :
   - o we can not create object of that class
   - o must inherit into your child class

2) using with method :
   - o do not specify body part of the method
   - o your class must be also abstract
   - o must override your abstract method into your child class.

23) Inheritance: Single, Multilevel, Hierarchical

→

1.Single Inheritance:

- **only one parent having only one child**

Ex:

```
class Animal {
   void sound() {
      System.out.println("Animal makes sound");
```

```
    }
}

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

2. Multilevel Inheritance

- single inheritance having one another child

Ex:

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

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

class Puppy extends Dog {
    void weep() {
        System.out.println("Puppy weeps");
    }
```

```
   }
```

3. Hierarchical Inheritance:

- one parent having 2 or more child

Ex:

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

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

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

24) Method Overriding and Dynamic Method Dispatch

→

1.Method Overriding:

- **A child class provides a new version of a method from the parent class.**
- **Method name, return type, and parameters must be same.**

Ex:

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

class Dog extends Animal {
  void sound() {
     System.out.println("Dog barks");
  }
}
```

2.Dynamic Method Dispatch:

When a parent class reference is used to call an overridden method in child class at runtime.

Ex:

```
    Animal a = new Dog();
    a.sound();
```

25) Constructor Types (Default, Parameterized)

→

1.Default Constructor

- Has no parameters
- Created automatically if not written

Ex:

```
   class Car {
Car() {
    System.out.println("Default constructor");
    }
 }
```

2. Parameterized Constructor

- ○ Has parameters to pass values when creating object

Ex:

```
           class Car {
       Car(String model) {
      System.out.println("Model: " + model);
   }
}
```

26) Copy Constructor (Emulated in Java)

→ Java doesn't have a built-in copy constructor but we can create one manually to copy data from one object to another.

Ex:

```
class Car {
    String model;


    Car(String model) {
        this.model = model;
    }
    Car(Car c) {
        this.model = c.model;
    }
}

public class Main {
    public static void main(String[] args) {
        Car c1 = new Car("Honda");
        Car c2 = new Car(c1);

        System.out.println(c2.model);
    }
}
```

27) Constructor Overloading

→

- Multiple constructors in a class with different parameters.
- Helps create objects in different ways.

Ex:

```
class Car {
    Car() {
        System.out.println("Default constructor");
    }

    Car(String model) {
        System.out.println("Model: " + model);
    }
    Car(String model, int year) {
        System.out.println(model + " - " + year);
    }
}
```

28) Object Life Cycle and Garbage Collection

→ Object Life Cycle in Java

1.Object Creation:

- Done using the new keyword.
- Example: Student s = new Student();

2. Object Usage:

- The object is used to call methods or access fields.
- Example: s.getName();

3. Object Becomes Unreachable:

- When no reference points to the object.
- Example: s = null;

4. Object is Eligible for Garbage Collection

Garbage Collection in Java

1. Java has an automatic Garbage Collector (GC).
2. It frees memory by removing unreachable objects.
3. Helps in memory management and avoids memory leaks.
4. You can request GC with:

Ex: System.gc();

29) One-Dimensional and Multidimensional Arrays

→ One-Dimensional Array:

at a time only one loop will be use

A **single row** of elements.

Ex:

int[] nums = {1, 2, 3, 4};

Multidimensional Array

loop with in loop will be used

An array of arrays (like a table).

Ex:2-D Array

```
int[][] matrix = {
  {1, 2},
  {3, 4}
};
```

30) String Handling in Java: String Class, StringBuffer, StringBuilder.

→ 1. String Class

Immutable (cannot be changed).

Stored in String pool.

Ex: String s = "Hello";

s = s.concat(" World");

2. StringBuffer

- Mutable
- Thread-safe
- Slower than StringBuilder.

Ex: StringBuffer sb = new StringBuffer("Hello");

sb.append(" World");

3. StringBuilder

- Mutable like StringBuffer.
- Not thread-safe (not synchronized).

- Faster than StringBuffer.

```
Ex: StringBuilder sb = new StringBuilder("Hello");

sb.append(" World");
```

31) Array of Objects

→Array of Objects:

- An array that holds references to objects.
- Used to store multiple objects of a class.

```
Ex: class Student {
  String name;
  Student(String n) {
    name = n;
  }
}


public class Test {
  public static void main(String[] args) {
    Student[] arr = new Student[3];
    arr[0] = new Student("John");
    arr[1] = new Student("Emma");
    arr[2] = new Student("Alex");


    for (Student s : arr) {
      System.out.println(s.name);
    }
  }
```

}

32) String Methods (length, charAt, substring, etc.)

→

Method	**Description**	**Example**
length()	Returns string length	"Java".length() → 4
charAt(int i)	Returns character at index i	"Java".charAt(1) → 'a'
substring(i)	Returns substring from index i	"Hello".substring(2) → "llo"
substring(i, j)	Returns substring from i to j-1	"Hello".substring(1, 4) → "ell"
toLowerCase()	Converts to lowercase	"JAVA".toLowerCase() → "java"
toUpperCase()	Converts to uppercase	"java".toUpperCase() → "JAVA"
equals(str)	Compares content	"a".equals("a") → true
equalsIgnoreCase(str)	Compares ignoring case	"a".equalsIgnoreCase("A") → true
contains(str)	Checks if string contains str	"hello".contains("el") → true
replace(a, b)	Replaces all a with b	"java".replace('a', 'o') → "jovo"
trim()	Removes leading/trailing spaces	" hello ".trim() → "hello"

33)Types of Inheritance in Java.

→ properties of parent class extends into child class.

Types of Inheritance in Java

1.Single Inheritance

- One subclass inherits from one superclass.
- Example: class A → class B extends A
- *Supported.*

2.Multilevel Inheritance

→

- A class inherits from a class, which inherits from another.
- Example: class A → class B extends A → class C extends B
- *Supported.*

3.Hierarchical Inheritance

- Multiple classes inherit from a single superclass.
- Example: class A → class B extends A, class C extends A
- *Supported*

4.Multiple Inheritance (with classes)

- One class inherits from multiple classes.
- *Not supported* directly (to avoid confusion/ambiguity).
- Supported via **interfaces**.

Benefits of Inheritance

1. Code Reusability – Write once, use many times.
2. Method Overriding – Change inherited method behavior.
3. Logical Structure – Organize classes in a hierarchy.
4. Easier Maintenance – Changes in parent class affect all child classes.
5. Extensibility – Easily add new features by extending existing classes.

34) Method Overriding

→ When a subclass provides its own version of a method that is already defined in its superclass.

Rules of Overriding

- Method name, return type, and parameters must be same.
- The method must be inherited from the parent class.
- Use @Override annotation (optional but recommended).
- Only instance methods can be overridden (not static or constructors).

```
Ex: class Animal {
  void sound() {
    System.out.println("Animal makes a sound");
  }
}


class Dog extends Animal {
  @Override
  void sound() {
    System.out.println("Dog barks");
  }
}
```

35) Dynamic Binding (Run-Time Polymorphism).

→ When the method call is resolved at runtime instead of compile time.

How It Works:

1. Achieved using method overriding.
2. Reference of parent class, but object of child class.
3. Java decides at runtime which method to call.

```
Ex: class Animal {
  void sound() {
    System.out.println("Animal sound");
  }
}

class Cat extends Animal {
  void sound() {
    System.out.println("Cat meows");
  }
}

public class Test {
  public static void main(String[] args) {
    Animal a = new Cat();
    a.sound();
  }
}
```

36) Super Keyword and Method Hiding.

→ Uses of super:

1.Access parent class method

Ex: super.methodName();

2. Access parent class variable

Ex: super.variableName;

3. Call parent class constructor

Ex: super();

Method Hiding

When a static method in subclass has the same signature as a static method in the superclass.

- It’s not overriding, it's hiding.
- Method call is resolved at compile-time, not runtime

Ex:

```
class A {
  static void show() {
    System.out.println("A's static method");
  }
}

class B extends A {
  static void show() {
    System.out.println("B's static method");
  }
}

public class Test {
  public static void main(String[] args) {
    A obj = new B();
    obj.show();
  }
```

}

37) Abstract Classes and Methods .

→An abstract class cannot be instantiated directly (you cannot create objects of it).

- It can have both abstract methods (without body) and concrete methods (with body).
- Abstract methods must be overridden in subclasses.
- Used to provide a common base with some shared code and some methods to be implemented by subclasses.

```
Ex: abstract class Animal {
  abstract void sound();
  void eat() {
    System.out.println("Eating");
  }
}
```

38) Interfaces: Multiple Inheritance in Java

→ An interface is a fully abstract type that contains only method declarations (before Java 8).

- Since Java 8, interfaces can have default and static methods with implementations.
- Interfaces allow multiple inheritance because a class can implement more than one interface, solving the "diamond problem" of multiple class inheritance.
- Used to define capabilities or contracts that classes agree to follow.

```
Ex: interface Flyable {
  void fly();
}
```

```
interface Swimmable {
  void swim();
}
```

39) **Implementing Multiple Interfaces**

→ A class can implement **multiple interfaces** by separating them with commas.

- The class must provide concrete implementations for **all abstract methods** declared in all interfaces.
- Enables combining different behaviors in a single class without the complications of multiple class inheritance.

```
Ex: class Bird implements Flyable, Swimmable {
  public void fly() {
    System.out.println("Bird is flying");
  }
  public void swim() {
    System.out.println("Bird is swimming");
  }
}
```

40) **Java Packages**

→ Built-in Packages: Provided by Java, e.g., java.lang, java.util.

User-Defined Packages: Created to organize related classes and avoid name conflicts.

Declared using:

package com.myapp.utils;

41) Access Modifiers

→

Modifier	**Same Class**	**Same Package**	**Subclass (diff package)**	**Anywhere**
private	Yes	No	No	No
default	Yes	Yes	No	No
protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes

42) **Importing Packages and Classpath**

→ Use import to access classes from other packages:

import java.util.Scanner;

Classpath tells JVM where to find classes.

Set classpath via command line or environment variables.

43) **Types of Exceptions**

→

1. **Checked Exceptions:**
   - Checked at **compile-time**.
   - Must be handled or declared with throws.
   - Example: IOException, SQLException.
2. **Unchecked Exceptions:**
   - Checked at **runtime** (not required to handle).
   - Subclass of RuntimeException.
   - Example: NullPointerException, ArithmeticException.

44) try, catch, finally, throw, throws

→ try — Block of code where exceptions may occur.

catch — Handles the exception thrown in try block.

finally — Executes always after try/catch (used for cleanup).

throw — Used to explicitly throw an exception.

throws — Declares exceptions a method may throw.

45) Custom Exception Classes

→ Create your own exceptions by extending Exception or RuntimeException.

Used for specific application error handling.

```
Ex: class MyException extends Exception {
  MyException(String message) {
    super(message);
  }
}
```

46) Introduction to Threads

→ A thread is the smallest unit of execution within a process. In Java, a program can have multiple threads running concurrently, enabling multitasking within a single program. Threads share the same memory space but execute independently, which improves application performance, especially on multi-core processors.

Key points:

- Java supports multithreading through the java.lang.Thread class and Runnable interface.
- Threads allow concurrent execution of two or more parts of a program.
- Useful for tasks like animation, I/O operations, and background computations.

47) **Creating Threads**

→ **1.Extending the Thread Class**

- Create a subclass of Thread and override the run() method, which contains the code executed by the thread.
- Create an instance of your subclass and call its start() method to begin execution.

```
 Ex:class MyThread extends Thread {
 public void run() {
   System.out.println("Thread running by extending Thread class");
 }
}


public class Test {
 public static void main(String[] args) {
   MyThread t1 = new MyThread();
   t1.start();
 }
}
```

2.Implementing the Runnable Interface

- Implement the Runnable interface and override the run() method.
- Create a Thread object by passing the Runnable object to its constructor.
- Call start() on the Thread object.

```
Ex: class MyRunnable implements Runnable {
 public void run() {
   System.out.println("Thread running by implementing Runnable");
 }
}
```

```
public class Test {
  public static void main(String[] args) {
    Thread t1 = new Thread(new MyRunnable());
    t1.start();
  }
}
```

48) **Thread Life Cycle**

→

State	**Description**
New	Thread object created but not started yet.
Runnable	Thread ready to run, waiting for CPU time.
Running	Thread is executing its run() method.
Blocked/Waiting	Thread waiting for a resource or condition (e.g., waiting for I/O, or waiting to acquire a lock).
Timed Waiting	Thread waiting for a specified period (e.g., sleep, join with timeout).
Terminated	Thread has finished execution or was stopped.

49) Synchronization and Inter-thread Communication

→ When multiple threads share resources (like variables or objects), synchronization is essential to prevent inconsistent or corrupt data caused by concurrent access.

a) Synchronization

- Achieved using the synchronized keyword in Java.

- Only one thread can execute a synchronized method or block at a time for a given object.
- Helps to avoid race conditions.

```
Ex: class Counter {
 private int count = 0;

 public synchronized void increment() {
  count++;
 }

 public int getCount() {
  return count;
 }
}
```

b) **Inter-thread Communication**

- Threads sometimes need to communicate, especially when one thread must wait for another to complete a task or signal it.
- Java provides methods like wait(), notify(), and notifyAll() for this purpose.
- These methods must be called from within synchronized blocks.

Ex:

```
class Message {
 private String message;
 private boolean hasMessage = false;
```

```
  public synchronized void write(String msg) throws InterruptedException {
    while (hasMessage) wait();
    message = msg;
    hasMessage = true;
    notify();
  }

  public synchronized String read() throws InterruptedException {
    while (!hasMessage) wait();
    hasMessage = false;
    notify();
    return message;
  }
}
```

50) Introduction to File I/O (java.io package)

→

- File I/O (Input/Output) in Java allows programs to **read from and write to files**, making it possible to store and retrieve data permanently.
- Java provides the **java.io package**, which contains classes for file handling like File, FileReader, FileWriter, BufferedReader, BufferedWriter, ObjectInputStream, and ObjectOutputStream.
- File I/O operations can be **character-based** (using Reader and Writer) or **byte-based** (using InputStream and OutputStream).
- Common use cases include:

- Reading configuration files
- Writing logs

- Saving user data
- Processing text files
- Exception handling is important in File I/O (e.g., IOException) to avoid runtime errors like "file not found" or "read/write failed".

```
Ex: File file = new File("example.txt");
if (file.createNewFile()) {
   System.out.println("File created.");
} else {
   System.out.println("File already exists.");
}
```

51) **FileReader and FileWriter**

→

- FileReader: Reads character data from a file.
- FileWriter: Writes character data to a file.

```
Ex: FileWriter fw = new FileWriter("data.txt");
fw.write("Hello");
fw.close();


FileReader fr = new FileReader("data.txt");
int i;
while ((i = fr.read()) != -1) {
  System.out.print((char)i);
}
fr.close();
```

52) **BufferedReader and BufferedWriter**

→ Used for fast reading/writing using buffer.

More efficient than FileReader/FileWriter.

```
Ex: BufferedWriter bw = new BufferedWriter(new FileWriter("data.txt"));
bw.write("Hello Buffered");
bw.close();

BufferedReader br = new BufferedReader(new FileReader("data.txt"));
String line;
while ((line = br.readLine()) != null) {
  System.out.println(line);
}
br.close();
```

53) Serialization and Deserialization

→ Serialization: Converting object into a byte stream.

Deserialization: Converting byte stream back to object.

Use ObjectOutputStream and ObjectInputStream.

Class must implement Serializable interface.

```
Ex: ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("obj.txt"));
out.writeObject(myObject);
out.close();

ObjectInputStream in = new ObjectInputStream(new FileInputStream("obj.txt"));
MyClass obj = (MyClass) in.readObject();
```

in.close();

54) Introduction to Collections Framework

→ The Collections Framework is a set of classes and interfaces in java.util used to store, retrieve, and manipulate data efficiently.

It provides ready-made data structures like List, Set, Map, and Queue.

Supports operations like sorting, searching, inserting, deleting, and iterating.

55) **Core Interfaces**

→

Interface	**Description**
List	**Ordered, allows duplicates (e.g., ArrayList, LinkedList)**
Set	**Unordered, no duplicates (e.g., HashSet, TreeSet)**
Map	**Stores key-value pairs (e.g., HashMap, TreeMap)**
Queue	**FIFO structure, used in scheduling tasks (e.g., PriorityQueue)**

56) Common Implementations

→

Class	**Description**
ArrayList	Dynamic array, fast access, slow insert/delete
LinkedList	Doubly linked list, fast insert/delete
HashSet	Unordered, fast lookup, no duplicates
TreeSet	Sorted set, no duplicates
HashMap	Key-value pairs, fast, unordered
TreeMap	Key-value pairs, sorted by keys

57) Iterators and ListIterators

→ **Iterator:** Used to **traverse collections** in forward direction.

```
 Ex:  Iterator<String> it = list.iterator();
while (it.hasNext()) {
  System.out.println(it.next());
}
```

ListIterator: Allows **forward and backward** traversal

```
Ex: ListIterator<String> lit = list.listIterator();
while (lit.hasNext()) { lit.next(); }
while (lit.hasPrevious()) { lit.previous(); }
```

58) Streams in Java (InputStream, OutputStream)

→ Streams are used to read and write data (especially bytes) in Java.

Located in java.io package.

Two main types:

- InputStream: Reads data from a source (e.g., file, keyboard).
- OutputStream: Writes data to a destination (e.g., file, console).

59) Reading and Writing Data Using Streams

→ InputStream Example (FileInputStream):

```
Ex: FileInputStream in = new FileInputStream("file.txt");
int i;
while ((i = in.read()) != -1) {
  System.out.print((char)i);
}
in.close();
```

OutputStream Example (FileOutputStream):

```
FileOutputStream out = new FileOutputStream("file.txt");
out.write("Hello".getBytes());
out.close();
```

60) Handling File I/O Operations

→ Streams help in reading/writing files in binary form (images, audio, text).

Always handle IOException using try-catch or throws.

Use FileInputStream and FileOutputStream for byte data.

Use FileReader and FileWriter for character data.