Enumeration (enum) is a special data type used to define a collection of named constant values.

- Introduced in java 5.
- Part of java.lang.Enum class.

Key features :

- Enum are implicitly final and static.
- Enum cannot extends other classes but can implement interfaces.
- They can have field, constructors, and methods.

`ordinal()` tells **at which place** (index) the enum constant appears.

```
// Creating an enum for
Days enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
                                                                }

public class Main {
    public static void main(String[]
        args) { Day today = Day.SUNDAY;

        // Using enum
        if (today == Day.SUNDAY) {
            System.out.println("Relax! It's
            Sunday.");
        } else {
            System.out.println("Workday!"); }}}
```

Annotations in java are metadata that provide additional information about program. Doesn't affect execution of program, but helps the complier, development tools, and frameworks process the code more efficiently. Widely used in frameworks like Spring, Hibernate and Junit.

Annotation	Meaning
`@Override`	Says that the method overrides a method from the parent.
`@Deprecated`	Says that the method/class is **old** and should not be used.
`@SuppressWarnings`	Tells the compiler to **ignore warnings**.

We also create our own annotation using @interface.

Use of Annotations :

- Improve code readability.
- Helps frameworks like spring and hibernate.
- Provide metadata for tools and compilers.

Exception in java is an event that disrupts the normal flow of program. It indicates that something went wrong during execution. Exception are Runtime error.

Complie time error : fix syntax or type issue; the compiler usually gives detailed messages about the problem.

Runtime error : handles with try-catch block & ensure correct input and logics are applied.

Logical error : the logic of the program to ensure that the correct approach is used to solve the problem.

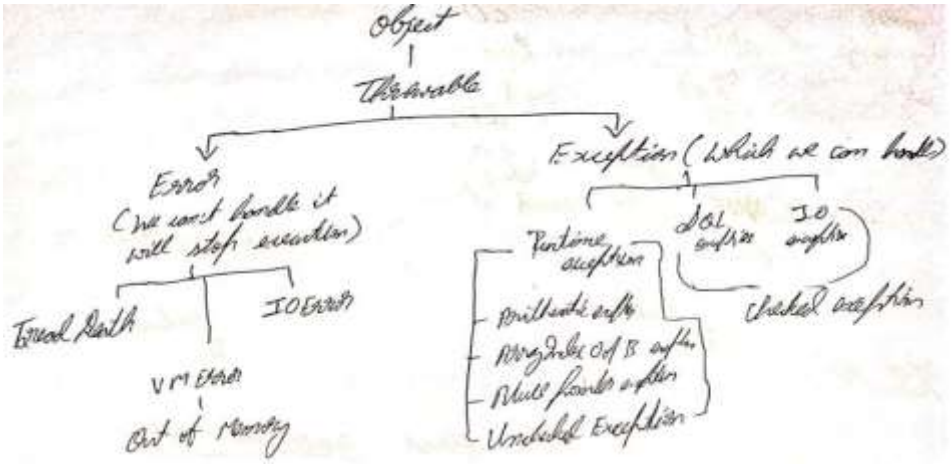
Why do we need Exception Handling?

- To avoid program crashes.
- To give user-friendly error messages.
- To maintain program flow even when errors happen.

Keyword	Meaning
`try`	Block to write code that might cause an exception.
`catch`	Block that handles the exception.
`finally`	Block that always executes (whether exception occurs or not).
`throw`	Used to manually throw an exception.
`throws`	Used to declare exceptions in method signature.

```
public class Example {
    public static void main(String[]
        args) { try {
            int a = 5 / 0;  // Problem: Division by zero
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide by zero!");
        } finally {
            System.out.println("This will always run.");  }}}
```

Catch block is executed only at the time of exception or else catch block will be skipped.



`throw` is used **inside a method** to manually throw an exception.

```
public class ThrowExample {
    public static void main(String[]
        args) { int age = 15;
        if (age < 18) {
            throw new ArithmeticException("Not eligible to vote");
        } else {
            System.out.println("Eligible to vote");     }}}
```

Here: We **manually threw** an `ArithmeticException` if the age is less than 18.

`throws` is used in the **method signature** to **declare** that the method might throw an exception.

```
import java.io.*;
public class ThrowsExample {
    public static void main(String[] args) throws IOException {
        // method call
        checkFile();
    }

    static void checkFile() throws IOException
        { throw new IOException("File not
        found");
                                                }
        }
```

Here:

- The `checkFile()` method **declares** that it might throw an `IOException`.
- So, the caller (`main`) either **handles** or **declares** it too.

Type	Meaning
Checked Exception	Must handle during coding.
Unchecked Exception	Happens during running program.
Error	Serious system problem (hard to handle).

1. Scanner (java.util)

- Easiest way to take input (keyboard, file, etc.).
- Methods like nextInt(), nextLine(), etc., are available.

```
import java.util.Scanner;
public class ScannerExample
{
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter your name:
        "); String name = sc.nextLine();
        System.out.println("Hello " + name);
                                            }
        }
```

---

2. InputStreamReader (java.io)

- Converts bytes from input (keyboard/file) into characters.
- Needs to be combined with BufferedReader for easy reading.

```
import java.io.*;
public class InputStreamReaderExample {
    public static void main(String[] args) throws IOException {
        InputStreamReader isr = new InputStreamReader(System.in);
        System.out.print("Enter a letter: ");
        char ch = (char) isr.read();  // Reads one
        character System.out.println("You entered: "
        + ch);                                  }}
```

---

3. BufferedReader (java.io)

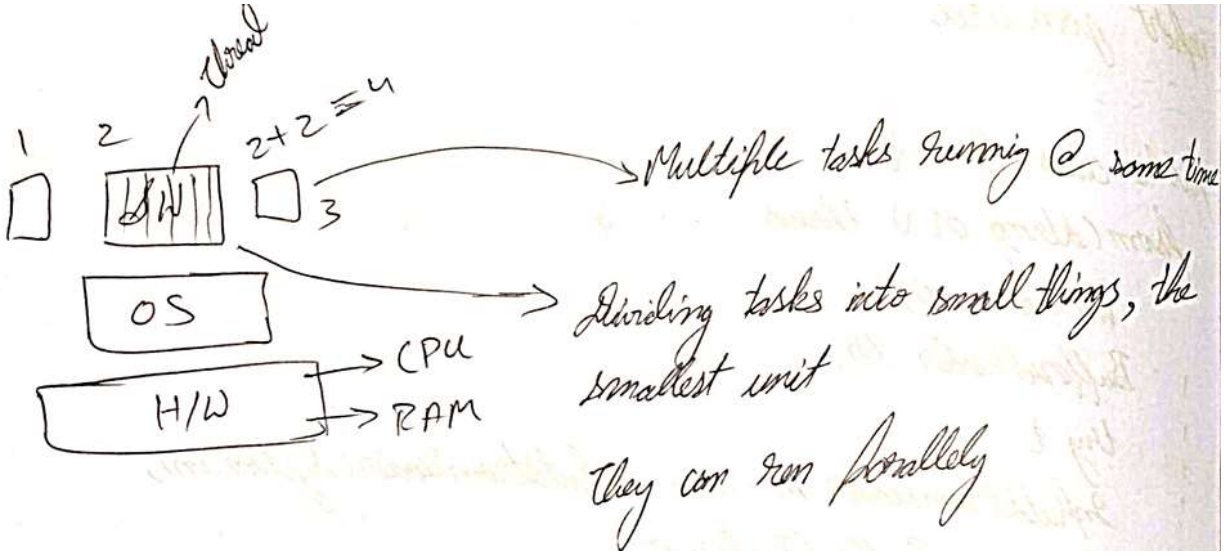
- Faster input reading.
- Reads entire lines or multiple characters at once.
- Works with InputStreamReader.

```
import java.io.*;
public class BufferedReaderExample {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        System.out.print("Enter your city: ");
        String city = br.readLine();  // Reads a
        full line System.out.println("City: " +
        city);                                  }}
```

Feature	Scanner	InputStreamReader	BufferedReader
Reads	Words, numbers, lines	Single characters	Whole lines
Easy to use?	Very easy	Little complex	Needs InputStreamReader
Speed	Moderate	Slow (alone)	Very fast
Used for	Simple input	Character conversion	Fast reading large input

Multithreading in java allows concurrent execution of 2 or more threads to maximize CPU utilization.

- It is implemented using the Thread class or Runnable interface.



A thread in java is the smallest unit of execution within a process.

- It runs independently and shares resources like memory with other threads in the same process.
- Java supports multithreading to enable parallel execution and improve performance.

```
class MyThread extends Thread {
    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println(i + " from " +
Thread.currentThread().getName());    }}}

public class MultiThread {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.start(); // Start first thread
        t2.start(); // Start second thread      }}
```

```
class MyRunnable implements Runnable {
    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println(i + " from " +
Thread.currentThread().getName());    }}}
```

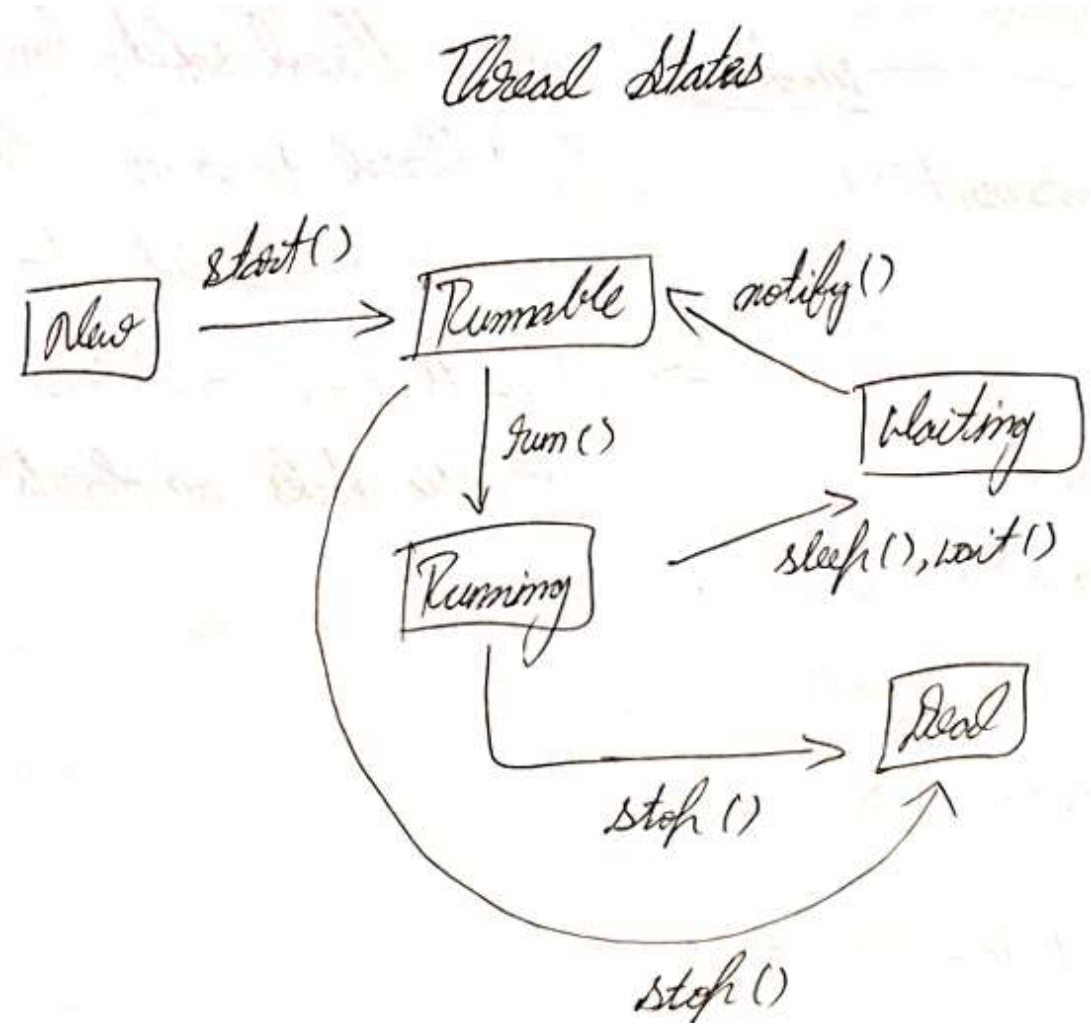
```
public class Multithreading {
    public static void main(String[] args) {
        MyRunnable myTask = new MyRunnable();  // Create object of class
        Thread t1 = new Thread(myTask);         // Wrap inside Thread object
        Thread t2 = new Thread(myTask);

        t1.start();  // Start first thread
        t2.start();  // Start second thread     }}
```

Runnable interface will not have thread method, so have to create separate object of it.

Scheduler is a part of OS or JVM that decides which thread to run based on factors like priority and thread state.

Thread priority is a value that determines the importance of the thread compared to other threads, influencing the order of execution.

getPriority() method in java used to retrieve the priority of thread. The range of priority goes from 1 to 10, 5 is default priority.



Aspect	Thread Class	Runnable Interface
Inheritance	Extends Thread	Implements Runnable
Flexibility	Can’t extend any other class	Can extend another class also

Collection API is a **set of classes and interfaces** in Java that are used to **store**, **manage**, and **manipulate** groups of **objects** easily.

- Collection = A container that holds multiple objects (like a basket holding fruits).
- API = Ready-made set of tools to use collections easily.
- Collection API gives us ready tools like `List`, `Set`, `Map` to store and manage many objects in an organized way.

Use Collection API

- To store **large amounts of data**.
- To **perform operations** like searching, sorting, inserting, deleting easily.
- To **avoid using arrays** when size is not fixed.

Interface	Meaning
`List`	Ordered collection, allows duplicates (e.g., ArrayList, LinkedList).
`Set`	No duplicate elements (e.g., HashSet, LinkedHashSet, TreeSet).
`Queue`	Elements processed in order (e.g., PriorityQueue, LinkedList).
`Map`	Stores **key-value pairs** (e.g., HashMap, TreeMap).

Term	Meaning
Collection API	A **framework** (set of **interfaces and classes**) to **store and manage groups of objects** (like List, Set, Map).
Collection	It is a **root interface** of the Collection Framework. It represents a **group of objects**. (List, Set, and Queue extend Collection.)
Collections	It is a **utility class** in `java.util` package that provides **static methods** like **sort()**, **reverse()**, **min()**, etc., to work on collections.

- **Collection API** → Full system/framework.
- **Collection** → Root level interface (base structure).
- **Collections** → Helper class with ready-made **methods**.

List Example:

```
import java.util.*;
public class CollectionList {
    public static void main(String[] args) {
        List<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Apple"); // duplicate allowed

        System.out.println(fruits);
}}
```

Set Example:

```
import java.util.*;
public class CollectionSet {
    public static void main(String[] args) {
        Set<String> fruits = new HashSet<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Apple"); // duplicate ignored
        System.out.println(fruits);  // Output: [Apple, Banana] (order not
guaranteed)  }}
```

Queue Example:

```
import java.util.*;
public class CollectionQueue {
    public static void main(String[] args) {
        Queue<String> fruits = new LinkedList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Mango");

        System.out.println(fruits.poll()); // Output: Apple (removed first)
        System.out.println(fruits);         // Remaining: [Banana,
Mango]       }}
```

Map Example:

```
import java.util.*;
public class CollectionMap {
    public static void main(String[] args) {
        Map<Integer, String> fruits = new HashMap<>();
        fruits.put(1, "Apple");
        fruits.put(2, "Banana");
        fruits.put(3, "Mango");
        System.out.println(fruits);  // Output: {1=Apple, 2=Banana,
3=Mango}        }}
```

Stream API in Java is used to **process collections of data** (like List, Set) in a very **easy**, **fast**, and **declarative way** (like saying *what* to do, not *how* to do).

✅ It **helps in performing** operations like:

- Filtering
- Sorting
- Mapping (transforming)
- Reducing (combining values)

✅ Introduced in **Java 8**.

```
import java.util.*;
import java.util.stream.*;

public class StreamAPI {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Chetan", "Arun", "Divya", "Anil");

        names.stream()                  // Create a stream
             .filter(name -> name.startsWith("A"))  // Filter names starting
with A
             .sorted()                  // Sort the names
             .forEach(System.out::println);  // Print each name           }}
```

Method	Purpose
`filter()`	Select elements based on condition
`map()`	Transform each element
`sorted()`	Sort elements
`collect()`	Collect back to List, Set, etc.
`forEach()`	Perform action for each element