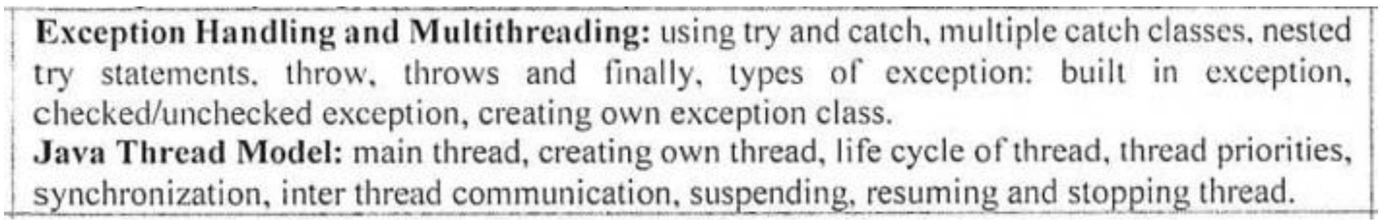
# UNIT 3



**Exception Handling and Multithreading:** using try and catch, multiple catch classes, nested try statements, throw, throws and finally, types of exception: built in exception, checked/unchecked exception, creating own exception class.
**Java Thread Model:** main thread, creating own thread, life cycle of thread, thread priorities, synchronization, inter thread communication, suspending, resuming and stopping thread.

## What is an Exception in Java?

In Java, exceptions are unexpected events or errors that disrupt the normal flow of a program's execution. Even if your code compiles successfully and looks error-free, some problems may only appear when the program runs. These are known as runtime errors, and Java handles them using exceptions. When an exception occurs, the program stops executing and displays an error message unless the exception is properly handled

## Example of Exception in Java

```
class ExceptionExample {
  public static void main(String args[]) {
    System.out.println("Welcome to ScholarHat");
    int a = 30;
    int b = 0;
    System.out.println(a / b);
    System.out.println("Welcome to the ScholarHat's Java Programming tutorial.");
    System.out.println("Enjoy your learning");
  }
}
```

In the above code, at the 4th line, an integer is divided by **0**, which is not possible, and the **JVM(Java Virtual Machine)** raises an exception. In this case, the programmer does not handle the **exception**, which will halt the program in between by throwing the exception, and the rest of the lines of code won't be executed.

## Output

```
Welcome to ScholarHat
```

**Exception in thread**

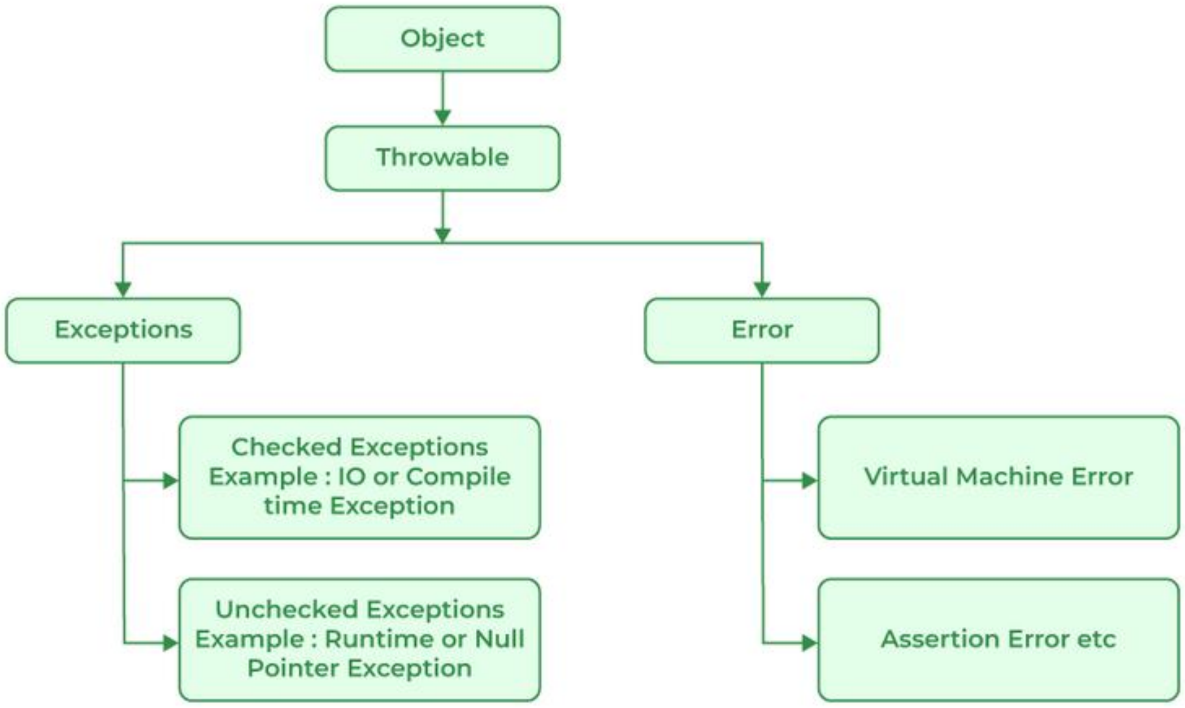
```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Main.main(Main.java:5)

"main" java.lang.ArithmeticException: / by zero
                at ExceptionExample.main(ExceptionExample.java:6)
```

## What is Exception Handling in Java?

Exception Handling is a way of handling errors that occur during runtime and compile time. It maintains your program flow despite runtime errors in the code and, thus, prevents unanticipated crashes. It facilitates troubleshooting by providing error details, cutting down on development time, and improving user happiness.

## Exception Hierarchy in Java


Object
Throwable
Exceptions
Error
Checked Exceptions Example : IO or Compile time Exception
Unchecked Exceptions Example : Runtime or Null Pointer Exception
Virtual Machine Error
Assertion Error etc

In Java, Exception and Error are direct subclasses of the Throwable class, which is the root of the exception hierarchy. An Exception represents conditions that a program should catch and handle, such as invalid input or file not found. An Error, however, indicates serious problems that occur in the Java Virtual

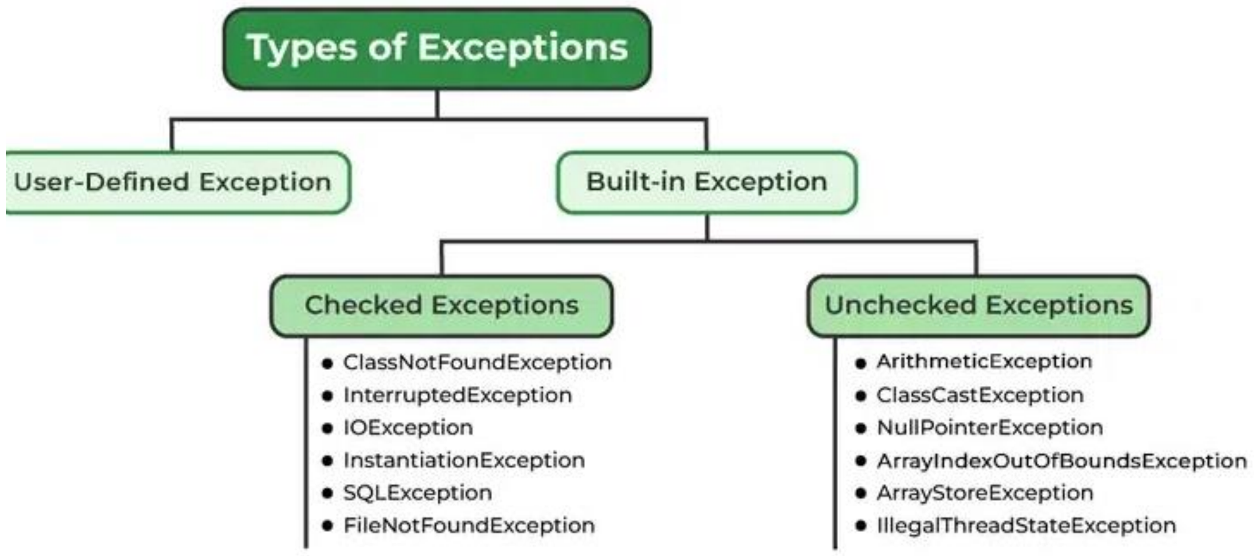
Machine (JVM), like Stack Over flow Error or Out Of Memory Error, and usually cannot be handled by the application code.

The hierarchy is divided into two branches:

- **Errors:** An error is a serious issue that occurs at runtime and is typically irrecoverable. It halts the normal execution of the program and cannot be handled by the programmer. Errors belong to the java. lang. Error class and include problems like Out Of Memory Error and Stack Over flow Error.
- **Exceptions:** Exceptions are events that a programmer can catch and handle within the code. When an exception occurs, Java creates an exception object that holds details such as the exception's name, message, and the program's state at the time. Exceptions allow the program to respond gracefully to unexpected situations.

We'll look at the types of exceptions below:

## Types of Exceptions in Java



There are mainly two types of exceptions: user-defined and built-in.

## Built-in Exceptions

Built-in exceptions are the exceptions that are already defined in Java libraries. They are part of the Java Exception Hierarchy and help handle common runtime errors, such as invalid input, division by zero, null references, and array bounds violations.
There are two types of built-in exceptions in Java:

### 1. Checked Exception

Checked exceptions are exceptions that are checked by the compiler at compile-time. This means if your code might throw a checked exception, Java will force you to either handle it using a try-catch block or declare it using the throws keyword in the method signature.

### 2. Unchecked Exception

Unchecked exceptions are exceptions that are not checked by the compiler at compile-time. These exceptions occur at runtime, and it's up to the programmer whether to handle them or not. They are also called runtime exceptions because the Java Virtual Machine (JVM) detects them while the program is running.

## User-Defined Exceptions

**User-defined exceptions** are also known as custom exceptions derived from the **Exception** class from **java. lang package(**Java package**)**. The user creates these exceptions according to different situations. Such exceptions are handled using five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
We'll learn how to use these keywords in the **Exception Handling Keywords in Java**section below. **Creating Own Exception Class**

- Users can create custom exceptions by extending the Exception class.

```
class MyException extends Exception {
   public MyException(String msg) {
      super(msg);
```

```
    }
}
```

| **Errors** | **Exceptions** |
|---|---|
| Belongs to the java. lang. Error class | Defined in java. lang Exception package |
| Errors are of Unchecked type | Exceptions can be both checked and unchecked. |
| Errors mainly occur during run-time. | Only the unchecked exceptions are encountered in run-time. |
| Errors are irrecoverable | Exceptions can be handled using exception-handling mechanisms |

## Why Do Exceptions Occur in Java?

Exceptions in Java occur due to unexpected events that prevent the program from running smoothly. These issues are usually not visible at the time of writing or compiling the code, but they show up during program execution (runtime).
Here are some common reasons why exceptions may occur in Java:

- **User's Invalid Input-** If a user enters input that the program is not expecting (e.g., entering a string when a number is expected), it can lead to exceptions like Number Format Exception.

- **Database Connection Error-** When the program cannot connect to the database (due to incorrect URL, credentials, or server issues), it may throw a SQLException.

- **System Failure-** Hardware issues such as disk failure or insufficient memory can cause unexpected exceptions during program execution.

- **Network Problems-** If your Java application depends on internet or server connections and the network is unavailable, exceptions like IO Exception or Socket Exception may occur.

- **Security Compromises-** When code tries to access a restricted resource without permission, a Security Exception can be thrown.

- **Errors in Code (Logical or Runtime)-** Mistakes in code, such as dividing by zero or accessing null references, are common causes of exceptions like Arithmetic Exception or Null Pointer Exception.

Physical Limitations- Running out of memory or file storage space can trigger exceptions such as Out Of Memory Error or IO Exception.

## Java Exception Handling Keywords

Java consists of **five keywords** to handle various kinds of custom exceptions. They are:

| Keyword | Description |
|---|---|
| try | The "try" keyword specifies an exception block. |
| catch | specifies the code block to be executed if an exception occurs in the try block. |
| finally | the **finally** block will always be executed whether an exception occurs or not |

| | |
|---|---|
| throw | the "throw" keyword triggers an exception |
| throws | The "throws" keyword declares an exception. |

## 1. try

- A try block consists of all the doubtful statements that may throw exceptions during program execution.
- A try block cannot work alone; it must be followed by at least one catch block or a finally block.
- If an exception occurs, the control immediately transfers from the try block to the appropriate catch block.
- The thrown exception object is caught by the catch block, which then handles the error as per the defined statements, allowing the program to continue running smoothly.

### Syntax

```
try
{
 //Doubtful Statements.
}
```

## 2. Catch

- The **catch** block handles the exception raised in the **try** block.
- The catch block or blocks follow every try block.
- The catch block catches the **thrown exception** as its parameter and executes the statements inside it.
- The declared exception must be the **parent class exception**, the generated exception type in the **exception class hierarchy**, or a **user-defined exception**.

### Syntax

```
try
 {
  // code
 }
 catch(Exception e)
 {
  // code to handle exceptions
 }
```

1. **single try-catch block**

```

class Main
{
  public static void main(String[] args)
  {
   try
   {
    // code that generate exception
    int divideByZero = 5 / 0;
    System.out.println("Rest of code in try block");
   }
   catch (ArithmeticException e) {
    System.out.println("ArithmeticException => " + e.getMessage());
   }
  }
 }
```

In the above code, we have put the "**int divideByZero=5/0**" in the try block because this statement must not be executed if the denominator is **0**. If the denominator is **0**, the statements after this statement in the try block are skipped. The catch block catches the thrown exception as its parameter and executes the statements inside it.

## Output

```
ArithmeticException => / by zero
```

### Multiple catch Blocks

We can use multiple catch statements for different kinds of exceptions that can occur from a single block of code in the try block.

## Syntax

```
try {
// code to check exceptions
}
catch (exception1) {
// code to handle the exception

}
catch (exception2) {
// code to handle the exception
}
.
.
.
catch (exception n) {
// code to handle the exception
}
```

## Example

```
public class MultipleCatchBlock {

 public static void main(String[] args) {

   try {
     int x[] = new int[5];
     x[5] = 40 / 0;
   } catch (ArithmeticException e) {
     System.out.println("Arithmetic Exception occurs");
   } catch (ArrayIndexOutOfBoundsException e) {
     System.out.println("ArrayIndexOutOfBounds Exception occurs");
   } catch (Exception e) {
     System.out.println("Parent Exception occurs");
   }
   System.out.println("The program ends here");
 }
}
```

Here, our program matches the **type of exception** that occurred in the **try block** with the catch statements. If the exception that occurred matches any of the usual catch statements, that particular catch block gets executed.

## Output

```
Arithmetic Exception occurs
The program ends here
```

1. **Nested try-catch**

   Here, we have a try-catch block inside a nested try block.

```
class NestingTry {
  public static void main(String args[]) {
    //main try-block
    try {
      //try-block2
      try {
        //try-block3
        try {
          int arr[] = {
            10,
            20,
            30,
            40
          };

          System.out.println(arr[10]);
        } catch (ArithmeticException e) {
          System.out.print("Arithmetic Exception");
          System.out.println(" handled in try-block3");
        }
      } catch (ArithmeticException e) {
        System.out.print("Arithmetic Exception");
        System.out.println(" handled in try-block2");
      }
    } catch (ArithmeticException e3) {
      System.out.print("Arithmetic Exception");
      System.out.println(" handled in main try-block");
    } catch (ArrayIndexOutOfBoundsException e4) {
      System.out.print("ArrayIndexOutOfBoundsException");
      System.out.println(" handled in main try-block");
    } catch (Exception e5) {
      System.out.print("Exception");
      System.out.println(" handled in main try-block");
    }
  }
}
```

Run Code >>

In the above code, the **Array Index Out Of Bounds Exception** occurred in the grandchild **try-block3**. Since try-block3 is not handling this exception, the control then gets transferred to the parent **try-block2**. Since try-block2 also does not handle that **exception**, the control gets transferred to the main **try-block**, where it finds the appropriate catch block for an exception.

### Output

**ArrayIndexOutOfBoundsException handled in main try-block**

## 3. finally

The **finally block in Java** always executes even if there are no **exceptions**. This is an optional block. It is used to execute important statements such as closing statements, releasing resources, and releasing memory. There could be one final block for every try block. This finally block executes after the **try...catch block**.

### Syntax

```
try
 {
  //code
 }
 catch (ExceptionType1 e1)
 {
  // catch block
 }
 finally
 {
  // finally block always executes
 }
```

### Example of Java Exception Handling using finally block in Java Playground

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

```
  // code that generates exception
  int divideByZero = 5 / 0;
 }
 catch (ArithmeticException e)
 {
  System.out.println("ArithmeticException => " + e.getMessage());
 }
 finally
 {
  System.out.println("This is the finally block");
 }
}
}
```

Run Code >>

| **final** | **finally** | **finalize** |
|---|---|---|
| final is a keyword and **access modifier**, which is used to apply restrictions on a class, method, or variable. | finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not. | finalize is the method in Java that is used to perform clean-up processing just before an object is garbage collected. |
| The final keyword is used with the classes, methods, and variables. | Finally, the block is always related to the try-catch block in exception handling. | finalize() method is used with the objects. |
| It is used with variables, methods, and classes. | It is with the **try-catch block** in exception handling. | Used with objects |

|  |  |  |
|---|---|---|
| Once declared the final variable becomes constant and can't be modified. A sub-class can neither override a final method nor can the final class be inherited. | **finally** block cleans up all the resources used in the try block | **finalize** method performs the cleaning concerning the object before its destruction |
| final method is executed only when we call it | finally block executes as soon as the execution of the try-catch block is completed without depending on the exception | finalize method is executed just before the object is destroyed |

In this Java example, trying to divide by zero results in an **Arithmetic Exception** that is caught and accompanied by an error message. The "**finally**" block also always runs, printing "**This is the finally block**" whether or not an exception was raised.

## Output

```
ArithmeticException => / by zero
This is the finally block
```

## 4. throw

- The throw keyword is used to explicitly throw a checked or an **unchecked exception**.
- The exception that is thrown needs to be of type Throwable or a subclass of Throwable.
- We can also define our own set of conditions for which we can throw an exception explicitly using the **throw keyword**.

- The program's execution flow stops immediately after the throw statement is executed, and the nearest try block is checked to see if it has a catch statement that matches the type of exception.

## Syntax

```
throw new exception_class("error message");
```

## Example of Exception Handling using Java throw

```
class ThrowExample {
  // Method to check if a number is negative
  public static void checkNumber(int number) {
    if (number < 0) {
      // Throwing an IllegalArgumentException if the number is negative
      throw new IllegalArgumentException("Number cannot be negative");
    } else {
      System.out.println("Number is " + number);
    }
  }

  public static void main(String[] args) {
    try {
      // Trying to check a negative number
      checkNumber(-5);
    } catch (IllegalArgumentException e) {
      // Handling the thrown exception
      System.out.println("Caught an exception: " + e.getMessage());
    }
  }
}
```

Run Code >>

In the main method, the exception is captured and handled using a **try-catch block**, showing the exception message if the input is negative. This Java class, **ThrowExample**, contains a method check Number that throws an **Illegal Argument Exception** if the input number is negative.

## Output

```
Caught an exception: Number cannot be negative
```

## 5. throws

The **throws keyword** is used in the method signature to indicate that a **method in Java** can throw particular exceptions. This notifies the method that it must manage or propagate these exceptions to the caller.

```
import java.io.IOException;

public class ThrowsExample {

  public static void main(String[] args) {
    try {
      methodWithException();
    } catch (IOException e) {
      System.out.println("Caught IOException: " + e.getMessage());
    }
  }

  public static void methodWithException() throws IOException {
    // Simulate an IOException
    throw new IOException("This is an IOException");
  }
}
```

Run Code >>

In the above code, the method methodWithException is declared with throws IOException, indicating that it may throw an IOException. The catch block catches the IOException in the main method.

### Output

```
Caught IOException: This is an IOException
```

| throw | throws |
|---|---|
| The **throw keyword** is used to explicitly throw an exception inside any block of code or function in the program. | Java throws keyword is used in method or function signature to declare an **exception** that the method may throw while execution of code |

|  |  |
|---|---|
| throw keyword can be used to throw both checked and unchecked exceptions | throws keyword can be used only with checked exceptions. |
| throw is used within the method. | throws is used within the method signature |
| **Syntax:** throw new exception_class("**error message**"); | **Syntax: void method()** throws **ArithmeticException** |
| We can throw only one exception at a time | We can declare multiple exceptions using the throws keyword that the method can throw |

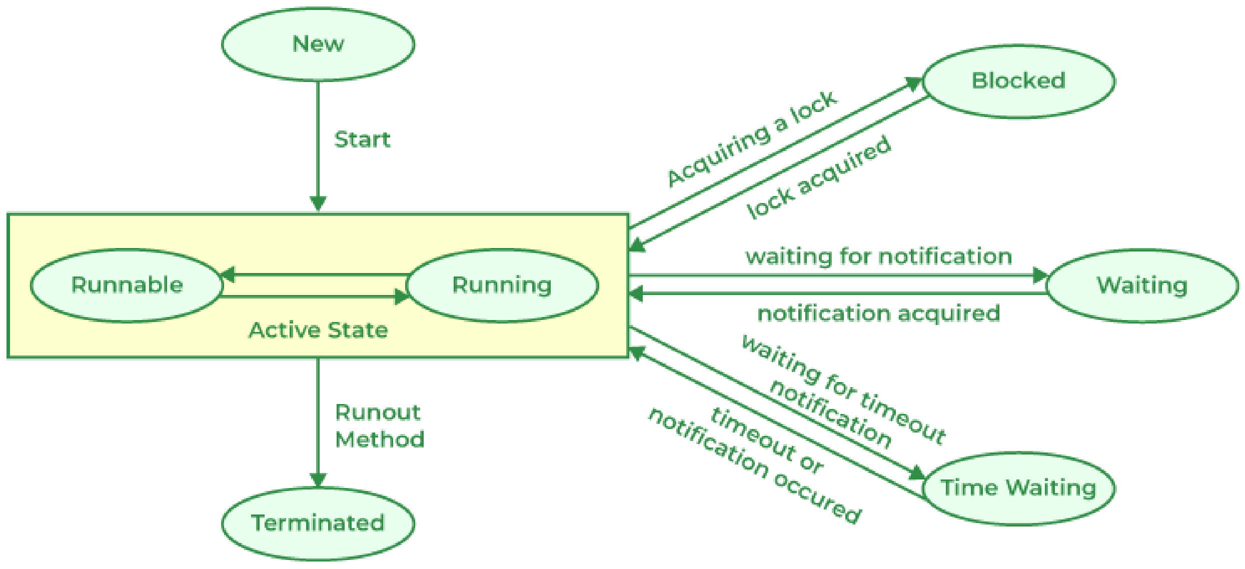
## Java Threads

A Java thread is the smallest unit of execution within a program. It is a lightweight subprocess that runs independently but shares the same memory space of the process, allowing multiple tasks to execute concurrently.
**For example:** In MS Word, one thread formats the document while another takes user input. Multithreading also keeps applications responsive, since other threads can continue running even if one gets stuck.

## Life Cycle of Thread

During its lifetime, a thread transitions through several states, they are:

1. New State
2. Active State
3. Waiting/Blocked State
4. Timed Waiting State
5. Terminated State



## Working of Thread States

**1. New State**: By default, a thread will be in a new state,  in this state, code has not yet started execution.

**2. Active State:** When a thread calls the start() method, it enters the Active state, which has two sub-states:

- **Runnable State**: The thread is ready to run but is waiting for the Thread Scheduler to give it CPU time. Multiple runnable threads share CPU time in small slices.
- **Running State:** When the scheduler assigns CPU to a runnable thread, it moves to the Running state. After its time slice ends, it goes back to the Runnable state, waiting for the next chance to run.

**3. Waiting/Blocked State:** a thread is temporarily inactive, it may be in the Waiting or Blocked state:

- **Waiting:** If thread T1 needs to use a camera but thread T2 is already using it, T1 waits until T2 finishes.
- **Blocked:** If two threads try to use the same resource at the same time, one may be blocked until the other releases it.

When multiple threads are waiting or blocked, the Thread Scheduler decides which one gets CPU based on priority.

**4. Timed Waiting State:** Sometimes threads may face starvation if one thread keeps using the CPU for a long time while others keep waiting.

**For example**: If T1 is doing a long important task, T2 may wait indefinitely. To avoid this, Java provides the Timed Waiting state, where methods like sleep() allow a thread to pause only for a fixed time. After the time expires, the thread gets a chance to run.

**5. Terminated State:** A thread enters the Terminated state when its task is finished or it is explicitly stopped. In this state, the thread is dead and cannot be restarted. If you try to call start() on a terminated thread, it will throw an exception.

## Create Threads in Java

We can create threads in java using two ways

- Extending Thread Class
- Implementing a Runnable interface

### 1. By Extending Thread Class

Create a class that extends Thread. Override the run() method, this is where you put the code that the thread should execute. Then create an object of your class and call the start() method. This will internally call run() in a new thread.

**Example:**

↔

```
class MyThread extends Thread
{
   // initiated run method for Thread
   public void run()
   {
      String str = "Thread Started Running...";
      System.out.println(str);
   }
}
```

↔

**Output**

```
Thread Started Running...
```

### 2. Using Runnable Interface

Create a class that implements Runnable. Override the run() method, this contains the code for the thread. Then create a Thread object, pass your Runnable object to it and call start().

**Example:**

```
class MyThread implements Runnable
{
   // Method to start Thread
   public void run()
   {
      String str = "Thread is Running Successfully";
      System.out.println(str);
   }

}
```

**Output**

```
Thread is Running Successfully
```

***Note:*** *Extend Thread when when you don't need to extend any other class. Implement Runnable when your class already extends another class (preferred in most cases).*

## Running Threads in Java

There are two methods used for running Threads in Java:

- run() Method in Java
- start() Method in Java

**Example:** Running a thread using start() Method

```
import java.io.*;
import java.util.*;

// Method 1 - Thread Class
class ThreadImpl extends Thread
{
    // Method to start Thread
    @Override
    public void run()
    {
        String str = "Thread Class Implementation Thread"
                    + " is Running Successfully";
      System.out.println(str);
    }
}

// Method 2 - Runnable Interface
class RunnableThread implements Runnable
{
     // Method to start Thread
     @Override
    public void run()
    {
        String str = "Runnable Interface Implementation Thread"
                    + " is Running Successfully";
      System.out.println(str);
    }

}

public class Geeks
{
    public static void main(String[] args)
    {
        // Method 1 - Thread Class
        ThreadImpl t1 = new ThreadImpl();
        t1.start();

        // Method 2 - Runnable Interface
      RunnableThread g2 = new RunnableThread();
      Thread t2 = new Thread(g2);
        t2.start();

        // Wait for both threads to finish before printing the final result
        try {
            // Ensures t1 finishes before proceeding
            t1.join();

            // Ensures t2 finishes before proceeding
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

## Output

```
Thread Class Implementation Thread is Running Successfully

Runnable Interface Implementation Thread is Running Successfully
```

The common mistake is starting a thread using run() instead of start() method.
*Thread myThread = new Thread(MyRunnable());*
*myThread.run(); //should be start()**;***
**Note:** We use start() to launch a new thread, which then calls the run() method in parallel. If we call run() directly, it works like a normal method call and no new thread is created.

## Threads Priorities

Priorities in Threads in Java is a concept where each thread has a priority in layman’s language one can say every object has priority here which is represented by numbers ranging from 1 to 10 and the constant defined can help to implement which are mentioned below.

| Constant | Description |
|---|---|
| public static int NORM_PRIORITY | Sets the default priority for the Thread. (Priority: 5) |
| public static int MIN_PRIORITY | Sets the Minimum Priority for the Thread. (Priority: 1) |
| public static int MAX_PRIORITY | Sets the Maximum Priority for the Thread. (Priority: 10) |

In case, we need to set Priority with a specific value(between 1-10) we will need some methods for it. Let us discuss how to get and set the priority of a thread in Java.

- **public final int getPriority():** java.lang.Thread.getPriority() method returns the priority of the given thread.
- **public final void setPriority(int newPriority):** java.lang.Thread.setPriority() method changes the priority of thread to the value newPriority. This method throws IllegalArgumentException if the value of parameter newPriority goes beyond the minimum(1) and maximum(10) limit.

**Example 1:** Setting and Getting Thread Priorities

```
import java.lang.*;

class Thread1 extends Thread {

   // run() method for the thread that is called as soon as start() is invoked for thread in
main()
   public void run()
   {
      System.out.println(Thread.currentThread().getName() + " is running with
priority "  + Thread.currentThread().getPriority());
   }

   public static void main(String[] args)
   {
      // Creating random threads with the help of above class
      Thread1 t1 = new Thread1();
      Thread1 t2 = new Thread1();
      Thread1 t3 = new Thread1();

      // Display the priority of above threads using getPriority() method
      System.out.println("t1 thread priority: " + t1.getPriority());
      System.out.println("t2 thread priority: " + t2.getPriority());
      System.out.println("t3 thread priority: " + t3.getPriority());

      // Setting priorities of above threads by passing integer arguments
      t1.setPriority(2);
      t2.setPriority(5);
      t3.setPriority(8);
```

```
        // Error will be thrown in this case t3.setPriority(21);

        // Last Execution as the Priority is low
        System.out.println("t1 thread priority: " + t1.getPriority());

        // Will be executed before t1 and after t3
        System.out.println("t2 thread priority: " + t2.getPriority());

        // First Execution as the Priority is High
        System.out.println("t3 thread priority: " + t3.getPriority());

        // Now Let us Demonstrate how it will work According to it's Priority
        t1.start();
        t2.start();
        t3.start();
    }
}
```

**Output:**

```
t1 thread priority: 5
t2 thread priority: 5
t3 thread priority: 5
t1 thread priority: 2
t2 thread priority: 5
t3 thread priority: 8
Thread-1 is running with priority 5
Thread-2 is running with priority 8
Thread-0 is running with priority 2
```

**Explanation:**

- Thread1 extends Thread and overrides run(), which prints the thread's name and priority.
- Threads have default priority 5, which can be changed between 1–10 using setPriority().
- The code sets t1=2, t2=5, t3=8 and prints their priorities.
- On calling start(), each thread runs concurrently; higher priority may get preference, but actual order depends on the OS thread scheduler.

## Synchronization

Synchronization in Java is a mechanism to control the access of multiple threads to shared resources. It is essential in a multithreaded environment to prevent thread interference and consistency problems. Synchronization ensures that only one thread can access the shared resource at a time, thus preserving the integrity of the data.

### Why Synchronization is Needed?

In a multithreaded environment, multiple threads may try to modify the same data simultaneously, leading to data inconsistency. Consider the following example without synchronization:

```java
class Counter {
    private int count = 0;

    public void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

class ThreadDemo extends Thread {
    Counter counter;
    ThreadDemo(Counter counter) {
        this.counter = counter;
    }

    public void run() {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Counter counter = new Counter();
        ThreadDemo t1 = new ThreadDemo(counter);
        ThreadDemo t2 = new ThreadDemo(counter);

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Final count: " + counter.getCount());
    }
}
```

In the above code, the `Counter` object is shared between two threads (`t1` and `t2`). Both threads increment the counter 1000 times. Ideally, the final count should be 2000. However, due to the lack of synchronization, the final count may be less than 2000 because the increment operation (`count++`) is not atomic and can be interrupted.

## How to Use Synchronization in Java?

Java provides various mechanisms to handle synchronization:

### 1. Synchronized Method:

Synchronize the entire method to ensure only one thread can execute it at a time.

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

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

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

Example: Synchronized Method

Here is a complete example demonstrating the use of synchronized methods:

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

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

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

class ThreadDemo extends Thread {
    Counter counter;

    ThreadDemo(Counter counter) {
        this.counter = counter;
    }

    public void run() {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Counter counter = new Counter();
        ThreadDemo t1 = new ThreadDemo(counter);
        ThreadDemo t2 = new ThreadDemo(counter);

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Final count: " + counter.getCount());
    }
}
```

Output:

```
Final count: 2000
```

In this example, the `increment()` method is synchronized. This ensures that only one thread can execute this method at a time, preserving the integrity of the `count` variable.

**2. Synchronized Block:**

Synchronize a block of code instead of the entire method, providing more control and efficiency.

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

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

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

**3. Static Synchronization:**

Synchronize static methods to ensure only one thread can execute them for the class, not the instance.

```
class Counter {
    private static int count = 0;

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

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

## 4. Locks:

Use `java.util.concurrent.locks.Lock` for more sophisticated thread synchronization.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class Counter {
    private int count = 0;
    private final Lock lock = new ReentrantLock();

    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }

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

## Inter-thread Communication in Java

Inter-thread Communication in Java allows synchronized threads to communicate with each other. Java provides `wait()`, `notify()`, and `notifyAll()` methods to facilitate this communication.

### Example of Inter-thread Communication

```
class SharedResource {
    private boolean ready = false;

    synchronized void produce() {
        try {
            while (ready) {
                wait();
            }
            System.out.println("Producing...");
            ready = true;
            notify();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    synchronized void consume() {
        try {
            while (!ready) {
                wait();
            }
            System.out.println("Consuming...");
            ready = false;
            notify();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class InterThreadCommunicationExample {
    public static void main(String[] args) {
        SharedResource resource = new SharedResource();

        Thread producer = new Thread(resource::produce);
        Thread consumer = new Thread(resource::consume);

        producer.start();
        consumer.start();
    }
}
```

This code demonstrates inter-thread communication in Java using a shared resource. It includes two main operations: producing and consuming. These operations are synchronized to ensure proper coordination between threads.

## Thread Control Methods in Java

Thread control methods are used to **pause, resume, stop, or control execution** of threads.

### 1. suspend() – (Deprecated)

- Used to **pause** a thread temporarily.
- When a thread is suspended, it **remains in suspended state until resumed**.
- **Problem**: If a thread suspends while holding a lock, other threads waiting for that lock are blocked forever → **Deadlock**.

**Example:**

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

```
            try { Thread.sleep(500); } catch (Exception e) {}
        }
    }
}

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

        try {
            Thread.sleep(1000);
            t.suspend();  // ❌Not recommended
            System.out.println("Thread suspended...");
            Thread.sleep(2000);
            t.resume();   // resume the thread
            System.out.println("Thread resumed...");
        } catch (Exception e) {}
    }
}
```

Deprecated since **Java 1.2**.
Better alternative: Use a **flag variable** (boolean) to pause and resume threads safely.

---

## 2. resume () – (Deprecated)

- Used to **restart a suspended thread**.
- Must be used with suspend().
- Problem: Can lead to **deadlocks** and **uncontrollable thread behavior**.

## 3. stop() – (Deprecated)

- Used to **terminate a thread immediately**.
- Problem: If a thread is stopped while updating shared resources → may leave data in **inconsistent state**.

**Example:**

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println("Running: " + i);
            try { Thread.sleep(500); } catch (Exception e) {}
        }
    }
}

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

        try {
            Thread.sleep(2000);
            t.stop();  // ❌Unsafe method
            System.out.println("Thread stopped...");
        } catch (Exception e) {}
    }
}
```

Deprecated because it can **kill a thread abruptly** without cleanup.
Better alternative: Use a **flag variable** or interrupt() method.