Multirasking:

Multitasking allows several activites to occur concurrently on the computer:

Process based multitasking

Thread based multitasking

Process Based Multitasking:

Allow process(i.e programs) to run concurrently on the computer

Eg:Running ms paint while work with word processor

Thread Based Multitasking:

All parts(feature) of same program to run concurrently on the computer

Eg:Ms word this printing and formatting text at the same time

Typing with autocorrection of word spell

|  |  |
| --- | --- |
| Thread  Two thread share same address space  Context switching between is usually less expensive than process  Cost of communication is relatively low  Thread is lightweight | Process |

Why MultiThreading?

Building of code base require more time in the time we do many task .

We won’t wait for particular task to complete and start the next task we do it paralley

In singe threaded environment ,only one task at a time can be performed

Cpu cycles are wasted for eg: when waiting for user input

Multitasking allow idle cpu time to put to good use

Threads:

A thread is an independent sequential path of execution within a program

When we want to create thread in main we use this multi threading

Many threads can run concurrenty within a program

At runtime threads in a program exist in a common memory space therforre they share both the code and data

Lightweight compared to process

The main thread:

When a standalone application is run,a user thread is automatically created execute main() methd in application.This thread is called main thread

IF no other thread are spawned the program terminates when main() methonf finishes excuting

All other threads called child thread spawned from main thread

The run time distinguish between user threads and daemon threads

**2. What is a Daemon Thread?**

* A **daemon thread** is a **background service thread**.
* It provides support to user threads (like garbage collection, background cleanup).
* The JVM **does NOT wait** for daemon threads. If **only daemon threads** are left, the JVM exits.

For example u create four user thread in main and one daemon thread and main terminates means the other foru user thread is excute so still alive when thet complete the jvm exists irrespective of daemon thread is running

**4. JVM Shutdown Rule**

The JVM **continues running as long as at least one user thread is alive**.

✅ Even if there are **100 daemon threads**, if **1 user thread is running**, JVM **won't exit**.

🚫 Daemon threads **don't keep the JVM alive**.

**3. setDaemon(true) Method**

This method **marks a thread as a daemon**:

Thread t = new Thread();

t.setDaemon(true); // Must be called before t.start()

⚠️ If you try to call setDaemon(true) **after** starting the thread, it throws IllegalThreadStateException.

These are the **two most common ways** to create a thread in Java.

 **Extending Thread class**

 **Implementing Runnable interface**

// MyTask.java

class Thread1 extends Thread {

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.println("Value of i: " + i + " | Thread: " + Thread.currentThread().getName());

}

}

}

// Main.java

public class Main {

public static void main(String[] args) {

Sysem.out.println(“Statre”);

Thread1 t1 = new Thread1();

T1.start();

System.out.println(“end”);

}

}

Start

End

Value of i: 1 | Thread: main

Value of i: 2 | Thread: main

Value of i: 3 | Thread: main

Value of i: 4 | Thread: main

Value of i: 5 | Thread: main

public class Main {

public static void main(String[] args) {

Sysem.out.println(“Statre”);

MyTask t1 = new MyTask();

T1.setDaemon(true);

T1.start();

System.out.println(“end”);

}

}

Output:

Start

End // as there is no other thread running main thread completed only demon thres jvm ignone

**start() vs run() in Java**

| **Feature** | **start()** | **run()** |
| --- | --- | --- |
| **Purpose** | Starts a new thread | Executes the code inside run() in current thread |
| **Thread Creation** | ✅ Yes, creates a new thread | ❌ No new thread — just a method call |
| **Defined in** | Thread class | Defined in Runnable interface |
| **Calls internally** | run() method | Your thread logic |
| **Concurrency** | Code runs in parallel (true multithreading) | Code runs sequentially in main thread |
| **Usage** | Always use this to start a thread | Don’t use directly for thread |

Thread Name:

 **Pass a name to a thread**,

 **Overload the constructor** in a custom thread class.

class MyTask extends Thread {

// Default constructor

public MyTask() {

super(); // calls Thread()

}

// Overloaded constructor with thread name

public MyTask(String name) {

super(name); // sets the thread name

}

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.println("Running " + i + " in thread: " + Thread.currentThread().getName());

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

}

}

}

public class Main {

public static void main(String[] args) {

System.out.println("Start");

MyTask t1 = new MyTask(); // uses default constructor

MyTask t2 = new MyTask("Worker-Thread"); // uses overloaded constructor

t1.start(); // Name will be something like "Thread-0"

t2.start(); // Name will be "Worker-Thread"

System.out.println("End");

}

}

Output:

Start

End

Running 1 in thread: Thread-0

Running 1 in thread: Worker-Thread

Running 2 in thread: Thread-0

Running 2 in thread: Worker-Thread

...

Impletenting in runnable interface

public class Main {

public static void main(String[] args) {

Thread t1 = new Thread(new thread2(), "MyRunnableThread");

t1.start();

}

}

class Thread2 implements Runnable {

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.println("Runnable Thread: " + Thread.currentThread().getName() + " -> " + i);

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

}

}

}

**What happens if you do?**

If you try to call .start() **a second time** on the same Thread object, you will get a **runtime exception**:

**✅ Output:**

Exception in thread "main" java.lang.IllegalThreadStateException

Why Do We Pass a Runnable Object to a Thread?

**Avoiding Inheritance Limitations**

* Java **only supports single inheritance** — a class can extend only one other class.
* If you want your class to have thread behavior *and* extend some other class (say, a GUI component or business logic class), you cannot extend Thread directly.

**Context: Where This Code Comes From**

You’re looking at the **run() method inside the Thread class**, which looks something like this (simplified from the JDK source):

public class Thread implements Runnable {

private Runnable target;

public Thread(Runnable target) {

this.target = target;

}

public void run() {

if (target != null) {

target.run();

}

}

}

**✅ Real-World Flow Summary**

1. You create a thread by passing a Runnable task:

Runnable r = new MyTask();

Thread t = new Thread(r); // target = r

1. You call:

t.start(); // creates a new OS-level thread and triggers run()

1. Inside that new thread, the JVM executes:

t.run(); // Thread.run() is called

1. That run() method checks:

if (target != null) // target is your Runnable object

1. Then it delegates:

target.run(); // your actual task logic is called and runs on the new thread

**🔍 Line-by-Line Deep Explanation**

**public void run() {**

* This is the **run method** of the Thread class.
* If you don't override it, this default version is used.
* If a Runnable was passed in the constructor, it gets stored in the target variable.

**if (target != null) {**

* This checks whether the Thread was constructed with a Runnable object.
* If not, target will be null, and nothing will happen (thread starts but does no work).

**target.run();**

* This is the **core delegation**.
* It **calls the run() method** of your Runnable object (your task).
* This is where **your logic executes**, inside the **new thread created by Thread.start()**.

**🔹 Analogy to Understand**

Imagine Thread is a **worker**, and Runnable is a **task sheet**.

* When you hire a worker (create a Thread), you hand them a task (pass a Runnable).
* When the worker starts (you call start()), they check:

"Do I have a task to do?"

* If they do, they perform the job (target.run()).

**🔸 What if Runnable is not passed?**

Then target == null. And the run() method does nothing — the thread will start and terminate instantly.

java

CopyEdit

Thread t = new Thread(); // No target

t.start(); // Thread starts but does nothing

Unless you **override Thread.run()** by subclassing Thread directly.

**🔹 Diagram Summary**

cpp

CopyEdit

Main Thread

↓

Thread t = new Thread(new MyTask()); // MyTask implements Runnable

↓

t.start() → JVM creates new thread

↓

t.run() → inside Thread class

↓

if (target != null) → true

↓

target.run() → calls MyTask's run() on the new thread

**✅ Why It Matters**

* This pattern allows Java to support:
  + **Decoupling** between thread management and task definition
  + **Thread reuse** with different jobs
  + **Executor frameworks** that handle thread pooling
  + **Lambda support** (since Runnable is a functional interface)

**🔚 Final Takeaway**

The line:

if (target != null) {

target.run();

}

...is the **heart of how Java’s Thread class executes custom task logic** passed via Runnable. It’s a delegation mechanism that makes the threading system flexible, reusable, and powerful.

Syncronization:

Threads share memory space,they can sshare resources\

However there are critical situations where it us desirable that only one thread at a time has access to shared resources

Avoid race condtion

A **race condition** occurs when:

**Two or more threads access a shared resource (like a variable or file) at the same time**, and the final outcome depends on the **order in which the threads execute**.

Because thread scheduling is handled by the JVM and OS, **you can't predict the order** — which leads to **unreliable and inconsistent behavior**.

**Locking in Java — Every Object Has a Monitor Lock**

In Java, **every object has a built-in lock**, also called a **monitor**.

So, when you write:

synchronized(obj) {

// critical section

}

What you're doing is:

* Acquiring the lock (monitor) of obj
* Only **one thread at a time** can hold the lock on that object
* Other threads must wait (block) until the lock is released

class MyStack {

private int[] stack;

private int top = -1;

private final int capacity;

private final Object lock = new Object(); // Custom lock object

public MyStack(int capacity) {

this.capacity = capacity;

this.stack = new int[capacity];

}

public void push(int value) {

synchronized (lock) { // Critical section for push

if (top == capacity - 1) {

System.out.println(Thread.currentThread().getName() + " → Stack Overflow");

return;

}

stack[++top] = value;

System.out.println(Thread.currentThread().getName() + " → Pushed: " + value);

}

}

public int pop() {

synchronized (lock) { // Critical section for pop

if (top == -1) {

System.out.println(Thread.currentThread().getName() + " → Stack Underflow");

return -1;

}

int value = stack[top--];

System.out.println(Thread.currentThread().getName() + " → Popped: " + value);

return value;

}

}

}

public class Main {

public static void main(String[] args) {

MyStack stack = new MyStack(5);

// Thread to push elements

Thread producer = new Thread(() -> {

for (int i = 1; i <= 6; i++) {

stack.push(i);

try { Thread.sleep(100); } catch (InterruptedException e) {}

}

}, "Producer");

// Thread to pop elements

Thread consumer = new Thread(() -> {

for (int i = 1; i <= 6; i++) {

stack.pop();

try { Thread.sleep(150); } catch (InterruptedException e) {}

}

}, "Consumer");

producer.start();

consumer.start();

}

}

**Critical Sections Explained**

* push() and pop() access and modify the shared data top and stack[].
* These methods are wrapped in:

synchronized (lock) { ... }

to ensure that only **one thread at a time** can perform push() or pop().

Look if u not use synchronized then pusp and pop is done by tw ghread simputaneously so when puh check the stack is empty and knows it have one element means while the elment in pop thread decrese the stack to -1 and make empty and te wejn u push st[-1]=value it show error

So we want push op to done first followed by pop op so we use synchronized with locks as object

**Locking is on the object, not on the method or thread**

* Only **one thread** can hold the lock on the lock object at a time.
* If **t1** holds the lock (executing synchronized block), then **other threads must wait** for the lock to be released before entering any synchronized block on the same lock.
* So **while t1 holds the lock**, no other thread can enter push() or pop().

**3. What happens when t1 is done?**

* When t1 **exits the synchronized block (or method)**, it **releases the lock**.
* Then **any other waiting thread** can acquire the lock and enter their synchronized blocks.

class MyStack {

private int[] stack;

private int top = -1;

private int capacity;

public MyStack(int capacity) {

this.capacity = capacity;

stack = new int[capacity];

}

public void push(int value) {

synchronized(this) { // locking on the current object

if (top == capacity - 1) {

System.out.println(Thread.currentThread().getName() + " → Stack Overflow");

return;

}

stack[++top] = value;

System.out.println(Thread.currentThread().getName() + " → Pushed: " + value);

}

}

public int pop() {

synchronized(this) { // locking on the current object

if (top == -1) {

System.out.println(Thread.currentThread().getName() + " → Stack Underflow");

return -1;

}

int val = stack[top--];

System.out.println(Thread.currentThread().getName() + " → Popped: " + val);

return val;

}

}

}

**What is synchronized(this)?**

* It is a **synchronization block** that locks on the **current object instance** (this).
* Only **one thread at a time** can hold the **monitor lock** on that particular object.
* If another thread tries to enter a synchronized(this) block (or a synchronized instance method) on the **same object**, it **must wait** until the lock is released.

**How does it work internally?**

* Every Java object has a **monitor (or intrinsic lock)**.
* When a thread enters synchronized(this), it requests the **monitor lock** on this object.
* If the monitor is **free**, the thread acquires it and proceeds inside the block.
* If the monitor is **already held** by another thread, the current thread is **blocked and waits**.
* When the thread **exits the synchronized block**, it **releases the monitor lock**, allowing other threads to acquire it.

**Why use this as the lock?**

* this refers to the **current instance** of the class.
* Locking on this means **only one thread can execute synchronized code on the same object instance** at a time.
* Different instances have different locks, so threads can concurrently access synchronized blocks on **different objects**.

**What is Daemon Thread in Java and explain their properties?**

Daemon thread is a low-priority thread that runs in the background to perform tasks such as garbage collection. It does possess certain specific properties as listed below:

* They can not prevent the JVM from exiting when all the user threads finish their execution.
* JVM terminates itself when all user threads finish their execution
* If JVM finds a running daemon thread, it terminates the thread and after that shutdown itself. JVM does not care whether the Daemon thread is running or not.
* It is an utmost low priority thread
* **What is the difference between the start() and run() method?**
* First**,** both methods are operated in general over the thread. So if we do use threadT1.start() then this method will look for the ***run() method*** to create a new thread. While in case of theadT1.run() method will be executed just likely the normal method by the "Main" thread without the creation of any new thread.

**✅ Can we Overload run() method in Java?**

Yes, you **can overload** the run() method, just like any other method in Java.

class MyTask extends Thread {

public void run() {

System.out.println("Default run() called");

}

public void run(int x) {

System.out.println("Overloaded run() with parameter: " + x);

}

} **BUT — here's the key point:**

* When you call start(), the JVM always calls the **no-argument run() method**.
* So the overloaded version like run(int x) will **not** be called automatically.

**How to call overloaded run()?**

Manually:

MyTask t = new MyTask();

t.start(); // Calls run()

t.run(5); // Calls overloaded version, but runs in main thread

❌ What if we do NOT override the run() method?

**🚫 Case 1: You extend Thread but don't override run()**

class MyTask extends Thread {

// No run() defined

}

public class Main {

public static void main(String[] args) {

MyTask t = new MyTask();

t.start(); // Nothing useful happens

}

} **Output:**

No output — because Thread class's default run() does nothing.

✅ The thread is created and starts, but it executes an **empty run()** — so it finishes immediately.

🚫 Case 2: You implement Runnable but forget to override run()

class MyTask implements Runnable {

// Forgot to define run()

}

public class Main {

public static void main(String[] args) {

Thread t = new Thread(new MyTask());

t.start(); // Error at runtime

}

} **Result:**

* You'll get a **compilation error** unless you mark MyTask as an abstract class.
* If run() is not defined and MyTask is concrete, it will complain because Runnable is a **functional interface** with an abstract method run().
* **Can we Override the start() method?**
* Even if we override the start() method in the custom class then no initializations will be carried on by the Thread class for us. The run() method is also not called and even a new thread is also not created.

**✅ Can you override start()?**

Yes. The start() method is **not final**, so it **can** be overridden in a subclass of Thread.

class MyThread extends Thread {

@Override

public void start() {

System.out.println("Custom start()");

super.start(); // still required to start a new thread

}

@Override

public void run() {

System.out.println("run() executing in: " + Thread.currentThread().getName());

}

} **⚠️ But be careful!**

If you override start() and **forget to call super.start()**, then:

* The thread will **not be created**.
* run() will execute in the **main thread**, just like a regular method call.

**❌ Wrong Example:**

class MyThread extends Thread {

@Override

public void start() {

System.out.println("Custom start() without calling super.start()");

run(); // Just calls run directly, no new thread is started

}

@Override

public void run() {

System.out.println("run() executing in: " + Thread.currentThread().getName());

}

} Custom start() without calling super.start()

run() executing in: main

👉 run() ran on the **main thread**, not a new one — **this breaks the purpose of multithreading**!

**What is volatile?**

volatile is a **keyword in Java** used to mark a variable so that:

1. **Its value is always read from main memory**, not from a thread’s cache.
2. **Changes made by one thread are immediately visible to other threads**.

**⚖️ High-Level Difference**

| **Feature** | **Exception** | **Error** |
| --- | --- | --- |
| **Definition** | Conditions your application **can handle** | Serious problems your app **should not handle** |
| **Type of** | Subclass of Throwable | Subclass of Throwable |
| **Recoverable?** | ✅ Yes, usually | ❌ No, usually |
| **Examples** | IOException, NullPointerException | OutOfMemoryError, StackOverflowError |
| **Use in app** | You are expected to **catch** or handle it | You are **not expected** to catch it |

Throwable

/ \

Error Exception

/ \

Checked Unchecked (RuntimeException)

**What is an exception?**

Exception is an abnormal condition which occurs during the execution of a program and disrupts normal flow of a program. This exception must be handled properly. If it is not handled, program will be terminated abruptly.

**2) How the exceptions are handled in Java? OR Explain exception handling mechanism in Java?**

Exceptions in Java are handled using try, catch and finally blocks.

try block : The code or set of statements which are to be monitored for exception are kept in this block.

catch block : This block catches the exceptions occurred in the try block.

finally block : This block is always executed whether exception is occurred in the try block or not and occurred exception is caught in the catch block or not.

**3) What is the difference between error and exception in Java?**

Errors are mainly caused by the environment in which an application is running. For example, OutOfMemoryError happens when JVM runs out of memory. Where as exceptions are mainly caused by the application itself. For example, NullPointerException occurs when an application tries to access null object.

**4) Can we keep other statements in between try, catch and finally blocks?**

No. We shouldn’t write any other statements in between try, catch and finally blocks.

try {

int a = 10 / 0;

}

System.out.println("Hello"); // ❌ Not allowed here

catch (ArithmeticException e) {

System.out.println("Exception caught");

}

finally {

System.out.println("Finally block");

}

**❗ Compile-time Error:**

error: 'catch' without 'try'

error: 'finally' without 'try'

**❌ Can we write only a try block without a catch or finally block?**

**No**, you **cannot write a try block alone**.  
Java **requires** that a try block must be followed by either:

* **at least one catch block**, or
* a **finally block**, or
* **both**.

🔴 Example — Invalid Code:

public class Test {

public static void main(String[] args) {

try {

int a = 10 / 0;

}

// No catch or finally

}

} **🧨 Compile-Time Error:**

error: 'try' without 'catch', 'finally' or resource declarations

✅ Valid Variations:

✅ 1. try + catch

try {

int a = 10 / 0;

} catch (ArithmeticException e) {

System.out.println("Caught exception");

}

**✅ 2. try + finally**

try {

int a = 10 / 0;

} finally {

System.out.println("This will always run");

}

**This will always run**

**Exception in thread "main" java.lang.ArithmeticException: / by zero**

**at YourClassName.main(YourClassName.java:line\_number)**

**✅ 3. try + catch + finally**

try {

int a = 10 / 0;

} catch (ArithmeticException e) {

System.out.println("Handled");

} finally {

System.out.println("Cleanup done");

}

**🧠 Why this rule?**

Because Java’s design enforces that **exceptions must be handled** or **cleanup must be ensured**.  
A try block by itself has **no purpose** unless you're handling or finalizing something.

**6) There are three statements in a try block – statement1, statement2 and statement3. After that there is a catch block to catch the exceptions occurred in the try block. Assume that exception has occurred in statement2. Does statement3 get executed or not?**

No, statement3 is not executed. Once a try block throws an exception, remaining statements will not be executed. Control comes directly to catch block.

**🧠 What is Exception Propagation?**

**Exception propagation** means:

If an exception is **not caught** in the current method, it is **automatically passed ("propagated")** to the caller method — and so on — **until it is caught** or causes the program to terminate.

**🏗️ How It Works — Step-by-Step**

**🔁 Propagation Chain**

main() → methodA() → methodB() → methodC()

↑ throws exception here

* If methodC() throws an exception, and it **doesn’t handle it**, Java checks:
  1. Does methodB() catch it?
  2. If not, does methodA() catch it?
  3. If not, does main() catch it?
  4. If no one handles it → **program crashes** with a runtime error.

🔷 Example 1: No Catch Block → Program Crashes

public class Test {

public static void methodC() {

int x = 10 / 0; // ArithmeticException

}

public static void methodB() {

methodC(); // No try-catch

}

public static void methodA() {

methodB(); // No try-catch

}

public static void main(String[] args) {

methodA(); // No try-catch

}

} 🔥 Output: Exception in thread "main" java.lang.ArithmeticException: / by zero

at Test.methodC(Test.java:3)

at Test.methodB(Test.java:7)

at Test.methodA(Test.java:11)

at Test.main(Test.java:15)

JVM shows the **call stack** because the exception **was never caught**, so it **propagated all the way** up to main() and crashed.

✅ Example 2: Catching the Exception in main()

public class Test {

public static void methodC() {

int x = 10 / 0; // ArithmeticException

}

public static void methodB() {

methodC(); // No catch

}

public static void methodA() {

methodB(); // No catch

}

public static void main(String[] args) {

try {

methodA(); // Catch here

} catch (ArithmeticException e) {

System.out.println("Exception handled in main: " + e);

}

}

} **✅ Output:**

Exception handled in main: java.lang.ArithmeticException: / by zero

✅ Even though the error happened in methodC(), it was **handled in main()** — thanks to **propagation**.

✅ 1. **try inside try** (Nested try)

A screen shot of a computer program

AI-generated content may be incorrect.

Outer try start

Inner try

Inner catch: java.lang.ArithmeticException: / by zero

Outer try end

✅ 2. **Exception not caught in inner, but caught in outer**

A screen shot of a computer code

AI-generated content may be incorrect.

Outer try

Outer catch: java.lang.NullPointerException

**7) What is unreachable catch block error?**

When you are keeping multiple catch blocks, the order of catch blocks must be from most specific to general ones. i.e sub classes of Exception must come first and super classes later. If you keep super classes first and sub classes later, compiler will show unreachable catch block error.

A diagram of a computer

AI-generated content may be incorrect.

**9) What are run time exceptions in Java. Give example?**

The exceptions which occur at run time are called as run time exceptions. These exceptions are unknown to compiler. All sub classes of java.lang.RunTimeException and java.lang.Error are run time exceptions. These exceptions are unchecked type of exceptions. For example, NumberFormatException, NullPointerException, ClassCastException, ArrayIndexOutOfBoundException, StackOverflowError etc.

**10) What is OutOfMemoryError in Java?**

OutOfMemoryError is the sub class of java.lang.Error which occurs when JVM runs out of memory.

**11) what are checked and unchecked exceptions in java?**

Checked exceptions are the exceptions which are known to compiler. These exceptions are checked at compile time only. Hence the name checked exceptions. These exceptions are also called compile time exceptions. Because, these exceptions will be known during compile time itself.

Unchecked exceptions are those exceptions which are not at all known to compiler. These exceptions occur only at run time. These exceptions are also called as run time exceptions. All sub classes of java.lang.RunTimeException and java.lang.Error are unchecked exceptions.

**13) Can we keep the statements after finally block If the finally block is returning the control?**

No, it gives unreachable code error. Because, control is returning from the finally block itself. Compiler will not see the statements after it. That’s why it shows unreachable code error.

A computer screen shot of a program code

AI-generated content may be incorrect.

Try block

Finally block

**🧨 Compilation error if you uncomment "After finally":**

error: unreachable statement

Because:

* The finally block's return **takes control away**.
* The compiler **knows** the next line is **unreachable**.

**14) Does finally block get executed If either try or catch blocks are returning the control?**

Yes, finally block will be always executed no matter whether try or catch blocks are returning the control or not.

**✅ Yes, the finally block always executes, regardless of whether:**

* the try block finishes normally
* the catch block handles an exception
* **a return statement** is used inside try or catch

Because the **finally block is designed for cleanup operations** (e.g., closing files, releasing resources), Java guarantees its execution unless the **JVM itself shuts down** (e.g., via System.exit() or fatal error).

**✅ Example 1: return inside try**

A computer code on a black background

AI-generated content may be incorrect.

Output:

In try

In finally

10

evenn though return 10 is in try, the finally **still runs before the method returns**.

A computer screen shot of a program code

AI-generated content may be incorrect.

OUTPUT:

In catch

In finally

20

🔥 BUT: When is finally block **not executed**?

**❌ Only in these rare cases:**

1. **System.exit(0)** is called before finally:

try {

System.exit(0);

} finally {

System.out.println("Finally"); // ❌ NOT printed

}

1. JVM crashes or the power goes out 😅
2. Infinite loop or deadlock prevents finally from executing

**What is Re-throwing an exception in Java?**

Exceptions raised in the try block are handled in the catch block. If it is unable to handle that exception, it can re-throw that exception using throw keyword. It is called re-throwing an exception.

A computer code with text

AI-generated content may be incorrect.

**What is the use of throws keyword in Java?**

throws keyword is used to specify the exceptions that a particular method can throw. The syntax for using throws keyword is,

For Checked Exceptions — compile-time safety

**. In contrast: Unchecked exceptions don’t require throws**

Because unchecked exceptions are usually **programming errors** (like NullPointerException), Java does **not require** explicit declaration or handling.

A computer screen shot of a computer

AI-generated content may be incorrect.

[throws Keyword In Java - Java Concept Of The Day](https://javaconceptoftheday.com/throws-keyword-java/)

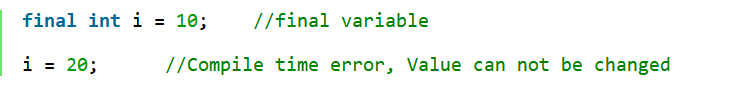
A screenshot of a computer program

AI-generated content may be incorrect.

**final keyword :**

final is a keyword which is used to make a variable or a method or a class as “**unchangeable**“. In simple terms,

A variable which is declared as final, it’s value can not be changed once it is initialized.



A method declared as final can not be overridden or modified in the sub class.

A computer code with text

AI-generated content may be incorrect.

A class declared as final can not be extended.

A close-up of a computer screen

AI-generated content may be incorrect.[final keyword in java - Java Concept Of The Day](https://javaconceptoftheday.com/final-keyword-in-java/)

1.final method can be overloaded and that overloaded method can be overridden in the sub class.

2.  When an array reference variable is declared as final, only variable itself is final but not the array elements.

 3.If the global variables are not initialized explicitly, they get default value at the time of object creation. But final global variables don’t get default value and they must be explicitly initialized at the time of object creation. Uninitialized final field is called **Blank Final Field**.

**Non-static final variable (instance-level)**

* Can be initialized:
  + **Inline** (at the point of declaration),
  + **Inside the constructor** (must be initialized in every constructor),
  + Or in an **instance initializer block**.

**Static final variable (class-level)**

* Can be initialized:
  + **Inline** (at the point of declaration),
  + Or inside a **static initializer block**.
* **finalize() Method :**
* **finalize() method** is a protected method of **java.lang.Object** class. It is inherited to every class you create in java. This method is called by garbage collector thread before an object is removed from the memory. finalize() method is used to perform some clean up operations on an object before it is removed from the memory.

A screenshot of a computer program

AI-generated content may be incorrect.

**finally Vs finalize() :**

But, there is one similarity between **finally block** and **finalize() method**. Both are used to close the resources used by the program. finally block is used to close the resources soon after their use. finalize() method is used to close the resources before an object is removed from the memory. That means if you use finalize() method to close the resources, they will remain open until an object,  which is using them, is garbage collected.

But, using finalize() method to close the resources is less recommended as it is not guaranteed that garbage collector will always call finalize() method on an object before it is removed from the memory. If it is not called, the resources will remain open. Therefore, it is always good to close the resources soon after their use using finally block.

A screenshot of a computer program

AI-generated content may be incorrect.

**What is StackOverflowError in Java?**

StackOverflowError is an error which is thrown by the JVM when stack overflows.

🔴 **What happens if the parent class does NOT have a method, but the child class has it — and we try to call that method using a parent reference?**

**✅ Answer:**

Java will give a **compile-time error**.

class Parent {

void show() {

System.out.println("Parent show()");

}

}

class Child extends Parent {

void show() {

System.out.println("Child show()");

}

void childOnlyMethod() {

System.out.println("Child-only method");

}

}

public class Test {

public static void main(String[] args) {

Parent p = new Child(); // 👈 Upcasting

p.show(); // ✅ This works (method exists in Parent and overridden in Child)

p.childOnlyMethod(); // ❌ Compile-time error!

}

}

**❌ Error:**

error: cannot find symbol

p.childOnlyMethod();

^

symbol: method childOnlyMethod()

location: variable p of type Parent

**❓Why this error?**

* p is declared as type Parent, so the compiler **only checks methods in Parent class**.
* Even though the object is Child, the **compiler doesn’t care** during compilation.
* It **only allows** calling methods **declared in the reference type** (Parent).
* Java uses the **reference type** for compile-time checking.

**✅ How to call the child method?**

* You need to **downcast** the reference:
* ((Child)p).childOnlyMethod(); // ✔️ Now works

**First: What is a Checked and Unchecked Exception?**

**🔷 Checked Exception:**

* **You MUST handle it** using try-catch or declare it using throws.
* The compiler **forces** you to handle it.
* Examples:
  + IOException
  + SQLException

👉 Think of it as: "Java says — *You better handle this problem!*"

**🔶 Unchecked Exception:**

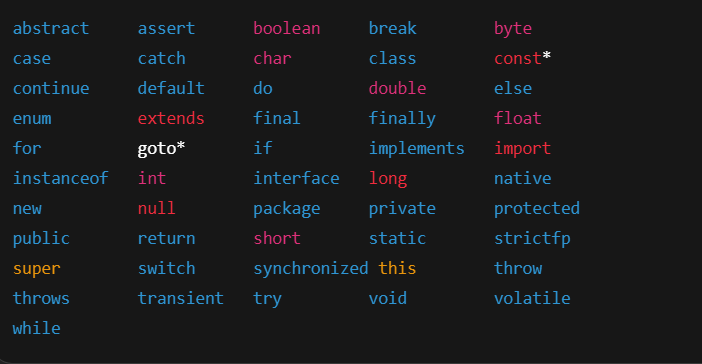
* **You DON'T have to handle it** — Java doesn’t force you.
* Happens because of programmer mistakes.
* Examples:
  + NullPointerException
  + ArithmeticException
  + ArrayIndexOutOfBoundsException

👉 Think of it as: "*If it happens, it crashes — but you weren’t forced to catch it.*"

**What is the use of printStackTrace() method?**

printStackTrace() method is used to print the detailed information about the exception occurred.

[30+ Java Exception Handling Interview Questions And Answers](https://javaconceptoftheday.com/java-exception-handling-interview-questions-and-answers/)

All 50 Java Keywords (As of Java 8+)**🟥 What are not keywords (but often confused)?**

These are **commonly misunderstood**:

| **Word** | **Is it Keyword?** | **What is it?** |
| --- | --- | --- |
| String | ❌ No | Class in java.lang |
| true | ❌ No (but special literal) | Boolean literal |
| false | ❌ No (but special literal) | Boolean literal |
| null | ❌ No (but special literal) | null reference value |
| System | ❌ No | Class in java.lang |
| out | ❌ No | Static field of System |
| println | ❌ No | Method in PrintStream class |

int class = 5; // ❌ Error: 'class' is a keyword

int String = 5; // ✅ Compiles (but confusing — don’t do this!)

**1) Is *String* a keyword in Java?**

No. *String* is not a keyword in Java. *String* is a final class in java.lang package which is used to represent the set of characters in Java.

**Is *String* a primitive type or derived type?**

*String* is a derived type.

**What are primitive types in Java?**

These are **basic built-in data types**, not derived from any class:

| **Primitive Type** | **Description** |
| --- | --- |
| int | Integer numbers |
| float | Decimal numbers |
| double | Bigger decimal numbers |
| char | Single character |
| byte | Small integer |
| short | Small integer |
| long | Big integer |
| boolean | true or false |

* These are **not objects**
* Stored in **stack memory**
* Don’t support methods

**🔶 Then what is a derived type (also called reference type)?**

**Derived types** (or **reference types**) are types **created from classes or interfaces**, like:

| **Derived Type Examples** |
| --- |
| String |
| ArrayList |
| int[] (arrays) |
| Custom classes (MyClass, Student, etc.) |

They are:

* **Objects** (instances of classes)
* Stored in **heap memory**
* Can have methods and properties
* Referred using **references (pointers)**

**🧠 String is derived because:**

* It’s an instance of java.lang.String class
* It has methods: .length(), .toUpperCase(), .substring(), etc.
* You create strings like objects:

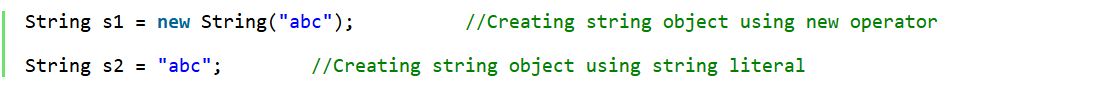
String s = new String("hello"); // Object form

Even when you do:

String s = "hello";

That’s just **syntactic sugar** for object creation — it's still a reference to a String object.

**In how many ways you can create string objects in Java?**

There are two ways to create string objects in Java. One is using *new* operator and another one is using string *literals*. The objects created using new operator are stored in the heap memory and objects created using string literals are stored in string constant pool. 

String Constant Pool:

String s1 = "hello"; // String literal

hello" is stored **once** in a special area of **heap memory** called the **String Constant Pool (SCP)**.

String s2 = "hello";

It **reuses** the same "hello" object — no new object is created.

Whenever you create a string object using string literal, JVM first checks the content of the object to be created. If there exist an object in the string constant pool with the same content, then it returns the reference of that object. It doesn’t create a new object. If the content is different from the existing objects then only it creates new object.

But What if You Use new String("hello")?

String s3 = new String("hello");

👉 **This creates two objects**:

1. One in the **String constant pool** ("hello" — if not already present)
2. Another in **heap memory** using new — a completely **new object**, even though it has the same content

String s1 = "hello";

String s2 = "hello"; // Uses same object as s1

String s3 = new String("hello"); // New object in heap

🧪 Comparisons:

System.out.println(s1 == s2); // true → same object from pool

System.out.println(s1 == s3); // false → s3 is a new heap object

System.out.println(s1.equals(s3)); // true → same content

☑️ Want to move heap string to pool manually?

String s4 = new String("hello").intern();

intern() tells JVM: "Check if the same string exists in the pool — if yes, return it; else add this to pool.

System.out.println(s1 == s4); // true

**What is special about string objects as compared to objects of other derived types?**

**One special thing about string objects is that you can create string objects without using new operator i.e using string literals. This is not possible with other derived types (except wrapper classes)**

**Why StringBuffer and StringBuilder classes are introduced in Java when there already exist String class to represent the set of characters?**

The objects of *String* class are immutable in nature. i.e you can’t modify them once they are created. If you try to modify them, a new object will be created with modified content. This may cause memory and performance issues if you are performing lots of string modifications in your code. To overcome these issues, *StingBuffer* and *StringBuilder* classes are introduced in Java.

**Which one will you prefer among “==” and equals() method to compare two string objects?**

*equals()* method because it compares two string objects based on their content.

use *“==”* operator, it checks only the references of two objects not their content

**Which class do you recommend among String, StringBuffer and StringBuilder classes if I want mutable and thread safe objects?**

StringBuffer

 **String** is **immutable** — once created, you cannot change its content.

 **StringBuffer** is **mutable** and **thread-safe** because all its methods are synchronized internally.

 **StringBuilder** is **mutable** but **not thread-safe**, so it's faster but unsafe in multi-threaded contexts.

**How do you convert given string to char array?**

Using *toCharArray()* method.

**I am performing lots of string concatenation and string modification in my code. which class among string, StringBuffer and StringBuilder improves the performance of my code. Remember I also want thread safe code?**

*StringBuffer* class gives better performance in this scenario. As *String* class is immutable, if you use this class, a new object will be created after every string concatenation or string modification. This will slow down the code. You can use *StringBuilder* also, but it is not thread safe. So, *StringBuffer* will be optimal choice here.

**What is string intern?**

String object in the string constant pool is called as *String Intern*. You can create an exact copy of heap memory string object in string constant pool. This process of creating an exact copy of heap memory string object in the string constant pool is called interning. *intern()* method is used for interning.

**What is the main difference between Java strings and C, C++ strings?**

In C and C++, strings are terminated with null character. But in Java, strings are not terminated with null character. Strings are treated as objects in Java.

**What do you think about string constant pool? Why they have provided this pool as we can store string objects in the heap memory itself?**

String constant pool increases the reusability of existing string objects. When you are creating a string object using string literal, JVM first checks string constant pool. If that object is available in string constant pool, it returns reference of that object rather than creating a new object. This will speed up your application as only reference is returned. And it also saves the memory as no two objects with same content are created.

**What does "\\s+" mean?**

* \s in regex means **any whitespace character** (space, tab, newline, etc.).
* + means **one or more occurrences** of the preceding element.
* In Java string literals, the backslash \ is an escape character, so to write a single backslash in the regex, you need to write \\.

we use *replaceAll()* method of *String* class to remove all white spaces (including tab also) from a string.  We pass the string **“\\s+”** to be replaced with an empty string **“”**

A computer code with blue text

AI-generated content may be incorrect.

**Output :**

Enter input string to be cleaned from white spaces…!  
OneSpace TwoSpaces  ThreeSpaces   FourSpaces    Tab        End  
Input String : OneSpace TwoSpaces  ThreeSpaces   FourSpaces    Tab        End  
Input String Without Spaces : OneSpaceTwoSpacesThreeSpacesFourSpacesTabEnd

1. *trim()* method trims the given string i.e it removes the white spaces at the beginning and at the end of a string, not between the words.

String to integer: ***Integer.parseInt()*** method and another one is ***Integer.valueOf()*** method

The main difference between *Integer.parseInt()* and *Integer.valueOf()* method is that *parseInt()* method returns primitive *int* where as *valueOf()* method returns *java.lang.Integer* object.

one is ***Integer.toString()*** method and another one is ***String.valueOf()*** method. Both these methods return string representation of the given integer.

[String To Integer And Integer To String Conversion In Java](https://javaconceptoftheday.com/string-to-integer-integer-to-string-conversion-in-java/)

[Java String Interview Questions And Answers](https://javaconceptoftheday.com/java-string-interview-questions-and-answers/)

**What are the fundamental principles of object oriented programming?**

a) [Inheritance](https://javaconceptoftheday.com/inheritance-in-java/)

b) [Abstraction](https://javaconceptoftheday.com/abstraction-in-java/)

c) [Polymorphism](https://javaconceptoftheday.com/polymorphism-in-java/)

d) [Encapsulation](http://www.tutorialspoint.com/java/java_encapsulation.htm)

Abstraction:

Abstraction is used to separate ideas from their implementation. Abstraction in java is used to define only ideas in one class so that the idea can be implemented by its sub classes according to their requirements

A screenshot of a computer program

AI-generated content may be incorrect.

Abstraction in java is implemented using Abstract classes and interfaces.

* It is not compulsory that abstract class must have abstract methods. It may or may not have abstract methods. But the class which has at least one abstract method must be declared as abstract.

[Abstraction In Java - Java Concept Of The Day](https://javaconceptoftheday.com/abstraction-in-java/)

**What is the parent class of all classes in Java?**

java.lang.Object class

**How do you restrict a member of a class from inheriting to it’s sub classes?**

By declaring that member as a private. Because, private members are not inherited to sub classes.

**Can a class extend itself?**

No, a class can not extend itself.

**Do constructors and initializers also inherited to sub classes?**

No, constructors and initializers (Static initializers and instance initializers) are not inherited to sub classes. But, they are executed while instantiating a sub class

**What happens if both, super class and sub class, have a field with same name?**

Super class field will be hidden in the sub class. You can access hidden super class field in sub class using super keyword.

**Do static members also inherited to sub classes?**

Yes, static members of a class are also inherited to sub classes.

**Why are static members inherited?**

1. **Static members belong to the class**, not to any particular instance.
2. When a subclass extends a superclass, it inherits the **class structure**, including static members.
3. The subclass can **access static fields and methods** defined in the superclass using either:
   * The subclass name, or
   * The superclass name (preferred).

[400+ Java Interview Questions And Answers](https://javaconceptoftheday.com/java-interview-questions-and-answers/)

**Can we overload main() method?**

Yes, we can overload main() method. A Java class can have any number of main() methods. But to run the Java class, class should have main() method with signature as public static void main(String[] args). If you do any modification to this signature, compilation will be successful. But, you can’t run the Java program. You will get run time error as main method not found.

**How many types of modifiers are there in Java?**

Two types of modifiers are there in Java. They are,

* Access Modifiers
* Non-access Modifiers

**What are access modifiers in Java?**

* [Access Modifiers In Java - Java Concept Of The Day](https://javaconceptoftheday.com/access-modifiers-in-java/)
* These are the modifiers which are used to restrict the visibility of a class or a field or a method or a constructor. Java supports 4 access modifiers.
* **a) private** : private fields or methods or constructors are visible within the class in which they are defined.
* **b) protected** : Protected members of a class are visible within the package but they can be inherited to sub classes outside the package.
* **c) public :** public members are visible everywhere.
* **d) default or No-access modifiers :** Members of a class which are defined with no access modifiers are visible within the package in which they are defined.

**What are non-access modifiers in Java?**

* These are the modifiers which are used to achieve the functionalities other than the accessibility. For example,
* **a) static :** This modifier is used to specify whether a member is a class member or an instance member.
* **b) final :** It is used to restrict the further modification of a class or a method or a field. (for more on final, [click here](https://javaconceptoftheday.com/final-keyword-in-java/)).
* **c) abstract :** abstract class or abstract method must be enhanced or modified further. (For more on abstract,  [click here](https://javaconceptoftheday.com/abstraction-in-java/)).
* **d) synchronized :** It is used to achieve thread safeness. Only one thread can execute a method or a block which is declared as synchronized at any given time. (
* **Can we use synchronized keyword with class?**
* No. synchronized keyword can be used either with a method or block.

**What is auto widening and explicit narrowing?**

* The data is implicitly casted from small sized primitive type to big sized primitive type. This is called auto-widening. i.e The data is automatically casted from byte to short, short to int, int to long, long to float and float to double..
* You have to explicitly cast the data from big sized primitive type to small sized primitive type.  i.e you have to explicitly convert the data from double to float, float to long, long to int, int to short and short to byte. This is called explicit narrowing.

[ClassCastException In Java - Java Concept Of The Day](https://javaconceptoftheday.com/classcastexception-in-java/) –need to see

**What is boxing and unboxing?**

Wrapping of primitive content into corresponding wrapper class object is called boxing. Unwrapping the wrapper class object into corresponding primitive content is called unboxing.

**What is the difference between auto-widening, auto-upcasting and auto-boxing?**

Auto-widening occurs when small sized primitive type is casted to big sized primitive type. Auto-upcasting occurs when sub class type is casted to super class type. Auto-boxing occurs when primitive type is casted to corresponding wrapper class.

**What is the method signature? What are the things it consists of?**

Method signature is used by the compiler to differentiate the methods. Method signature consist of three things.

* Method name
* Number of arguments
* Types of arguments

**Is it possible to have two methods in a class with same method signature but different return types?**

* No, compiler will give duplicate method error. Compiler checks only method signature for duplication not the return types. If two methods have same method signature, straight away it gives compile time error.

**What Happens Instead? Method Hiding**

If a subclass defines a static method **with the same name and signature** as one in its superclass:

* It **hides** the superclass method — it does **not override** it.
* The version that gets called is determined by the **reference type**, not the object type.

class Parent {

static void display() {

System.out.println("Parent static method");

}

}

class Child extends Parent {

static void display() {

System.out.println("Child static method");

}

}

public class Test {

public static void main(String[] args) {

Parent p = new Child();

p.display(); // Output: Parent static method

Child c = new Child();

c.display(); // Output: Child static method

}

}

**Can we override protected method of super class as public method in the sub class?**

Yes. You can increase the visibility of overriding methods but can’t reduce it.

**List Implementations**

| **Class** | **Thread-Safe** | **Maintains Order** | **Allows Duplicates** | **Performance** |
| --- | --- | --- | --- | --- |
| ArrayList | ❌ | ✅ | ✅ | Fast for random access, slow for insert/delete |
| LinkedList | ❌ | ✅ | ✅ | Fast for insert/delete at ends |
| Vector | ✅ | ✅ | ✅ | Legacy, synchronized |
| Stack | ✅ (extends Vector) | ✅ | ✅ | LIFO behavior |

**🔷 4. Set Implementations**

| **Class** | **Ordered?** | **Sorted?** | **Allows null?** | **Thread-safe** |
| --- | --- | --- | --- | --- |
| HashSet | ❌ | ❌ | One null | ❌ |
| LinkedHashSet | ✅ (insertion) | ❌ | One null | ❌ |
| TreeSet | ✅ (sorted) | ✅ | ❌ | ❌ |

**5. Map Implementations**

| **Class** | **Maintains Order** | **Null Keys** | **Null Values** | **Thread Safe** |
| --- | --- | --- | --- | --- |
| HashMap | ❌ | One | Multiple | ❌ |
| LinkedHashMap | ✅ (insertion) | One | Multiple | ❌ |
| TreeMap | ✅ (sorted) | ❌ | ✅ | ❌ |
| Hashtable | ❌ | ❌ | ❌ | ✅ |
| ConcurrentHashMap | ❌ | ❌ | ✅ | ✅ (partial locking) |

**Difference between Collection and Collections**

| **Feature** | **Collection** | **Collections** |
| --- | --- | --- |
| Type | **Interface** | **Utility class** |
| Package | java.util | java.util |
| Purpose | Root interface for data structure classes | Provides utility methods (sort, sync) |
| Common Methods | add(), remove(), size() | sort(), reverse(), shuffle(), etc. |
| Usage Example | Collection<String> list = new ArrayList<>(); | Collections.sort(list); |