

Exceptions in Java



explain about exception in java

Got it 👍 Let's go step by step and deeply explain **Exceptions in Java** with definitions, examples, real-world scenarios, types, handling strategies, and best practices.

◆ 1. What is an Exception in Java?

- An **Exception** in Java is an **unexpected event** that disrupts the normal flow of a program.
- It usually occurs during **runtime** (not compile-time).
- Java uses an **exception handling mechanism** to gracefully manage these errors instead of crashing the program.

👉 Exceptions are objects (instances of classes) that inherit from the **Throwable** class.

◆ 2. Exception Hierarchy in Java

```
pgsql

Throwable
└─ Error (serious problems, not handled usually)
    └─ Exception
        └─ Checked Exception (must be handled at compile time)
            └─ Unchecked Exception (runtime exceptions, optional to handle)
```

◆ 3. Types of Exceptions

(A) Checked Exceptions

- Known as **compile-time exceptions**.
- Compiler forces you to handle them with **try-catch** or **throws**.
- Example: **IOException**, **SQLException**, **FileNotFoundException**.

📌 Example:

```
java

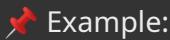
import java.io.*;

public class CheckedExample {
    public static void main(String[] args) {
        try {
            FileReader fr = new FileReader("test.txt"); // File may not exist
        } catch (FileNotFoundException e) {
```

```
        System.out.println("File not found!");
    }
}
```

(B) Unchecked Exceptions

- Known as **runtime exceptions**.
 - Compiler doesn't force handling.
 - Example: `NullPointerException`, `ArithmeticException`, `ArrayIndexOutOfBoundsException`.

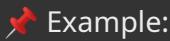


```
java

public class UncheckedExample {
    public static void main(String[] args) {
        int a = 10, b = 0;
        int result = a / b; // ArithmeticException
        System.out.println(result);
    }
}
```

(C) Errors

- Serious issues that applications should not try to handle.
 - Example: **OutOfMemoryError**, **StackOverflowError**.

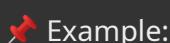


```
java

public class ErrorExample {
    public static void main(String[] args) {
        recursive(); // will cause StackOverflowError
    }
    public static void recursive() {
        recursive();
    }
}
```

◆ 4. Exception Handling Keywords

- **try** → Block of code that might throw an exception.
 - **catch** → Block that handles the exception.
 - **finally** → Always executes (used for cleanup).
 - **throw** → Used to explicitly throw an exception.
 - **throws** → Declares exceptions a method can throw.



```
java

public class ExceptionHandling {
    public static void main(String[] args) {
        try {
            int[] arr = {1, 2, 3};
            System.out.println(arr[5]); // risky code
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Array index out of range!");
        } finally {
            System.out.println("Finally block always runs.");
        }
    }
}
```

◆ 5. Custom Exceptions

You can create your own exceptions by extending **Exception** or **RuntimeException**.

📌 Example:

```
java

class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public class CustomExceptionExample {
    static void checkAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or above.");
        }
        System.out.println("Valid age!");
    }

    public static void main(String[] args) {
        try {
            checkAge(15);
        } catch (InvalidAgeException e) {
            System.out.println("Caught Exception: " + e.getMessage());
        }
    }
}
```

◆ 6. Real-World Usage of Exceptions

1. **File Handling** → Checking if file exists before reading.
2. **Database Connections** → Handling **SQLException** when database is unreachable.
3. **User Input** → Handling invalid inputs.
4. **Network Communication** → Handling **IOException** when server is down.

◆ 7. Differences: Checked vs Unchecked Exceptions

Feature	Checked Exception	Unchecked Exception
Compile-time checked?	Yes	No
Package	<code>java.io, java.sql</code>	<code>java.lang</code>
Must handle explicitly?	Yes	No
Example	<code>IOException</code>	<code>NullPointerException</code>

♦ 8. Best Practices for Exception Handling

- ✓ Catch only what you can handle.
- ✓ Always clean up resources (use `finally` or **try-with-resources**).
- ✓ Don't swallow exceptions (don't just `catch (Exception e) {}`).
- ✓ Create custom exceptions for business logic errors.
- ✓ Prefer **specific exceptions** over generic ones.

✓ In short:

Exception in Java is an object that represents an abnormal situation. It provides a way to handle errors gracefully and maintain smooth program execution.

Do you want me to also explain **try-with-resources** (modern Java way of handling exceptions with auto resource management) with an example?



difference between exception and error

Great question — let's clearly differentiate **Exception** and **Error** in Java with definition, examples, real-world meaning, and a comparison table.

♦ 1. Exception

• Definition:

An **Exception** is a problem that occurs during program execution and can **usually be recovered** by the program.

- They represent **conditions a program should anticipate and handle**.
- Part of `java.lang.Exception` class hierarchy.

📌 Examples

- `FileNotFoundException` → file not found
- `SQLException` → database connection issue
- `NullPointerException` → accessing null object
- `ArithmaticException` → divide by zero

📌 Real-world analogy:

Imagine you entered the wrong password — the system doesn't crash, it just shows an error message and asks again.

◆ 2. Error

- **Definition:**

An **Error** represents serious issues that occur in the **JVM (Java Virtual Machine)**, and **cannot usually be recovered** by the program.

- They indicate problems **beyond the control of the program**.
- Part of **java.lang.Error** class hierarchy.

📌 Examples

- **OutOfMemoryError** → JVM runs out of memory
- **StackOverflowError** → infinite recursion
- **VirtualMachineError** → JVM crashes

📌 Real-world analogy:

Imagine the computer itself loses power — your program cannot handle this; it just stops.

◆ 3. Key Differences Between Exception and Error

Feature	Exception	Error
Definition	Recoverable problems in program logic	Serious problems in JVM or environment
Belongs to	java.lang.Exception	java.lang.Error
Recoverable?	Yes, can be caught and handled	No, usually not recoverable
When occurs?	Due to bad code or input	Due to system/JVM failure
Handling required?	Yes, especially checked exceptions	No, usually should not be handled
Examples	IOException, NullPointerException	OutOfMemoryError, StackOverflowError
Program control	Can handle with try-catch	Program usually terminates

◆ 4. Code Illustration

📌 Exception Example

```
java

public class ExceptionDemo {
    public static void main(String[] args) {
        try {
            int a = 10 / 0; // ArithmeticException
        } catch (ArithmetiException e) {
            System.out.println("Handled Exception: " + e);
        }
    }
}
```

```
}
```

📌 Error Example

```
java

public class ErrorDemo {
    public static void main(String[] args) {
        try {
            recursive(); // causes StackOverflowError
        } catch (StackOverflowError e) {
            System.out.println("Can we catch it? Yes, but not recommended!");
        }
    }
    static void recursive() {
        recursive();
    }
}
```

👉 Notice:

- **Exceptions** → can be handled and program can continue.
- **Errors** → may be caught, but usually not advisable (program is unstable).

✓ In short:

- **Exceptions** = recoverable issues in your code (file not found, wrong input).
- **Errors** = serious JVM/system-level failures (out of memory, stack overflow).

Would you like me to also show a **real-world web app scenario** comparing where **Exceptions vs Errors** occur (e.g., handling missing user input vs JVM crash)?



explain throw and throws

Perfect 👍 let's break down **throw** and **throws** in Java in a detailed way with definitions, syntax, code examples, real-world analogies, and comparison.

◆ 1. **throw in Java**

- **Definition:**
throw is a **keyword** used to **explicitly throw a single exception object** from a method or block.
- Only **one exception** can be thrown at a time.
- The object thrown must be of type **Throwable** (i.e., **Exception** or **Error**).

✓ Syntax:

```
java
throw new ExceptionType("Error Message");
```

📌 Example (using throw):

```
java

public class ThrowExample {
    public static void main(String[] args) {
        int age = 15;
        if (age < 18) {
            throw new IllegalArgumentException("Age must be 18 or above.");
        }
        System.out.println("Eligible to vote!");
    }
}
```

👉 Here, we **explicitly throw** `IllegalArgumentException` when `age < 18`.

◆ 2. throws in Java

- **Definition:**
`throws` is used in the **method declaration** to specify which exceptions a method **can throw**.
- It tells the **caller of the method** to handle these exceptions.
- Multiple exceptions can be declared separated by commas.

✓ Syntax:

```
java

returnType methodName() throws ExceptionType1, ExceptionType2 {
    // method code
}
```

📌 Example (using throws):

```
java

import java.io.*;

public class ThrowsExample {
    public static void readFile() throws IOException {
        FileReader fr = new FileReader("test.txt"); // may throw IOException
        System.out.println("File opened successfully.");
    }

    public static void main(String[] args) {
        try {
            readFile(); // must handle IOException here
        } catch (IOException e) {
            System.out.println("Exception handled: " + e.getMessage());
        }
    }
}
```

👉 Here, the method **declares** it might throw **IOException**, and the **caller handles it**.

◆ 3. Real-World Analogy

- **throw** → Think of it like **raising a complaint** yourself ("I found a problem, here it is").
- **throws** → Think of it like a **disclaimer** ("This method might cause some issues, you must be prepared to handle them").

◆ 4. Key Differences Between **throw** and **throws**

Feature	throw	throws
Usage	Used inside method/block to throw exception	Used in method declaration to declare exceptions
Number of exceptions	Can throw only one exception at a time	Can declare multiple exceptions
Followed by	Followed by an exception object	Followed by exception class names
Position	Inside method body	After method signature
Example	throw new IOException("File not found");	void readFile() throws IOException

◆ 5. Combined Example (**throw + throws**)

```
java

class InvalidAgeException extends Exception {
    public InvalidAgeException(String msg) {
        super(msg);
    }
}

public class ThrowThrowsExample {
    // Method declares it may throw InvalidAgeException
    static void checkAge(int age) throws InvalidAgeException {
        if (age < 18) {
            // Actually throwing exception object
            throw new InvalidAgeException("Age must be 18 or above.");
        }
        System.out.println("Valid age, access granted!");
    }

    public static void main(String[] args) {
        try {
            checkAge(15);
        } catch (InvalidAgeException e) {
            System.out.println("Caught Exception: " + e.getMessage());
        }
    }
}
```

👉 Output: Caught Exception: Age must be 18 or above.

✓ In short:

- **throw** → actually throws the exception object.
- **throws** → declares what exceptions a method *can throw*, shifting responsibility to the caller.

Would you like me to also add a **diagram showing the flow of throw vs throws** (method declaring exception vs actual throwing)?



does any name can be used while throw

Good question

The short answer is: **No, you can't use just any name with throw.**

The object you throw must always be an **instance of a class that extends Throwable** (i.e., **Exception** or **Error**).

◆ 1. What throw Accepts

- **throw** requires an **object reference** of type **Throwable** or its subclasses.
- Example:

java

```
throw new NullPointerException("Null value found!");
throw new IOException("File not found!");
throw new MyCustomException("Business rule violated!");
```

Valid because all extend **Throwable**.

◆ 2. What You Cannot Throw

You cannot throw:

- Primitive values (**int**, **float**, etc.)
- Random objects not extending **Throwable** (like **String**, **Object**, **ArrayList**, etc.)

Example (Invalid):

java

```
throw new String("Error"); // ✗ Compilation error
throw 100; // ✗ Primitive type not allowed
throw new Object(); // ✗ Object is not Throwable
```

◆ 3. Rule Recap

- `throw` → **needs an object instance**
 - That object must be of type **Throwable** (checked/unchecked exception, error, or custom exception).
 - You can create your own class by extending **Exception** or **RuntimeException** and throw its object.
-

◆ 4. Example with Custom Exception

```
java

class MyCustomException extends Exception {
    public MyCustomException(String msg) {
        super(msg);
    }
}

public class ThrowDemo {
    public static void main(String[] args) {
        try {
            throw new MyCustomException("This is my custom exception!");
        } catch (MyCustomException e) {
            System.out.println("Caught: " + e.getMessage());
        }
    }
}
```

👉 Output: Caught: This is my custom exception!

✓ So: You can't use *any* random name or type with `throw`.
It must be an **exception object** (`Throwable` or subclass).

Do you want me to also explain why Java designers **restricted throw only to Throwable types** (instead of letting any object be thrown)?



why we need exception, let me explain with clear scenario

Excellent thought 🌟 — instead of just theory, let's **understand why we actually need exceptions in Java** with a **clear real-world scenario**.

◆ Why Do We Need Exceptions?

- Without exceptions, **any runtime error would crash the program immediately**.
- Exceptions provide a way to **detect, handle, and recover** from errors gracefully.
- They make code more **robust, maintainable, and user-friendly**.

◆ Scenario 1: File Reading Without Exceptions

Suppose you want to read a file that stores student details:

👉 *Code without exception handling:*

```
java

public class WithoutException {
    public static void main(String[] args) {
        java.io.FileReader fr = new java.io.FileReader("students.txt"); // risky
        System.out.println("File opened successfully!");
    }
}
```

👉 If `students.txt` does not exist:

- Program **crashes immediately** with `FileNotFoundException`.
- User sees a **confusing stack trace** instead of a friendly message.
- Program cannot continue execution.

◆ Scenario 2: File Reading With Exceptions

Now with exception handling:

👉 *Code with exception handling:*

```
java

import java.io.*;

public class WithException {
    public static void main(String[] args) {
        try {
            FileReader fr = new FileReader("students.txt");
            System.out.println("File opened successfully!");
        } catch (FileNotFoundException e) {
            System.out.println("⚠ Sorry, students.txt is missing. Please upload the file.");
        } finally {
            System.out.println("Program continues execution...");
        }
    }
}
```

👉 If `students.txt` does not exist:

- Program **does not crash**.
- User sees:

csharp

```
⚠ Sorry, students.txt is missing. Please upload the file.
Program continues execution...
```

- The app is still usable.

◆ Scenario 3: Real-World Example — Banking System

Imagine an **ATM withdrawal system**:

- You try to withdraw ₹10,000 but your account has only ₹5,000.
- **Without exception handling** → Program crashes with a runtime error. Machine stuck, card not returned! 😠
- **With exception handling** → Program throws `InsufficientBalanceException`, shows a message:

```
csharp
```

```
Transaction failed: Balance too low.  
Please try again with a smaller amount.
```

👉 ATM still works, and your card is safely ejected.

◆ Scenario 4: Web Application

In an online shopping app:

- User enters **wrong credit card number**.
- **Without exception handling** → Application crashes.
- **With exception handling** → Program catches `InvalidCardException`, shows:

```
sql
```

```
Payment failed. Please check your card number.
```

👉 User corrects input and continues shopping.

◆ Summary

We need exceptions because they:

- ✓ Prevent program crashes.
- ✓ Provide **user-friendly error messages**.
- ✓ Allow **graceful recovery** and continuation of execution.
- ✓ Help **separate error-handling logic** from normal logic.
- ✓ Improve **robustness** of real-world applications.

✓ In short:

Without exceptions → small errors = full program crash.

With exceptions → small errors = handled gracefully, program continues.

Do you want me to also show you a **flow diagram** comparing what happens when an exception occurs **with vs without exception handling**?



how to create custom exception

Absolutely! Let's go **step by step** to create and use **custom exceptions** in Java. I'll explain with **code examples, rules, and real-world scenarios**.

◆ 1. What is a Custom Exception?

- A **custom exception** is an exception **defined by the programmer** to represent specific error conditions in your application.
- Useful when **built-in exceptions** (`IOException`, `NullPointerException`, etc.) don't clearly describe the error.
- All custom exceptions must **inherit from `Exception`** (checked) or `RuntimeException` (unchecked).

◆ 2. Steps to Create a Custom Exception

Step 1: Create a class that extends `Exception` (checked) or `RuntimeException` (unchecked)

```
java

// Checked Exception Example
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message); // call Exception class constructor
    }
}

// Unchecked Exception Example
class NegativeAmountException extends RuntimeException {
    public NegativeAmountException(String message) {
        super(message);
    }
}
```

Step 2: Throw the exception in a method

```
java

public class CustomExceptionDemo {

    // Checked Exception method
    static void checkAge(int age) throws InvalidAgeException {
        if (age < 18) {
            throw new InvalidAgeException("Age must be 18 or above.");
        }
        System.out.println("Age is valid!");
    }

    // Unchecked Exception method
    static void withdraw(int amount) {
        if (amount < 0) {
            throw new NegativeAmountException("Amount cannot be negative!");
        }
        System.out.println("Withdrawal successful: " + amount);
    }
}
```

```

}
public static void main(String[] args) {
    // Handling checked exception
    try {
        checkAge(15);
    } catch (InvalidAgeException e) {
        System.out.println("Caught Exception: " + e.getMessage());
    }

    // Unchecked exception (optional handling)
    withdraw(-500); // Runtime exception, can be caught or left unhandled
}
}

```

Output:

```

php

Caught Exception: Age must be 18 or above.
Exception in thread "main" NegativeAmountException: Amount cannot be negative!

```

◆ 3. Key Points**1. Checked vs Unchecked**

- Checked → extend `Exception`, must handle using `try-catch` or `throws`.
- Unchecked → extend `RuntimeException`, handling is optional.

2. Constructor

- Usually define a constructor that calls `super(message)` to set the exception message.

3. Throwing

- Use `throw new YourCustomException("message")` whenever the error condition occurs.

◆ 4. Real-World Scenario**• Banking app:**

```

java

class InsufficientBalanceException extends Exception {
    public InsufficientBalanceException(String msg) {
        super(msg);
    }
}

java

void withdraw(double amount) throws InsufficientBalanceException {
    if (amount > balance) {
        throw new InsufficientBalanceException("Insufficient balance!");
    }
    balance -= amount;
}

```

- **Benefit:** The error clearly describes the **specific business rule violation**, not a generic exception.

✓ Summary

1. Create a class extending `Exception` or `RuntimeException`.
2. Add a constructor that calls `super(message)`.
3. Throw it using `throw` keyword when needed.
4. Handle checked exceptions with `try-catch` or declare using `throws`.

If you want, I can make a **diagram showing flow of a custom exception from method → throw → catch → finally**, which makes it super clear visually.

Do you want me to do that?



`super(msg);`

what does this means

Ah, good question! Let's break it down clearly.

◆ 1. What is `super(msg)?`

In Java, `super` is a **keyword** that refers to the **parent (super) class** of the current class.

When you write:

```
java
super(msg);
```

it means:

- Call the **constructor of the parent class**.
- Pass the argument `msg` to it.

◆ 2. Why we use `super(msg)` in custom exceptions?

When you create a custom exception:

```
java
class MyException extends Exception {
    public MyException(String msg) {
        super(msg); // calling Exception(String message) constructor
```

```
}
```

- **Exception** class has a constructor:

```
java

public Exception(String message) {
    // stores the error message inside the exception object
}
```

- By calling **super(msg)**, you **pass the error message to the parent Exception class**.
- This way, when you later do:

```
java

throw new MyException("Something went wrong!");
```

You can retrieve the message using:

```
java

catch (MyException e) {
    System.out.println(e.getMessage()); // prints: Something went wrong!
}
```

◆ 3. Real-World Analogy

Think of it like **forwarding a message**:

- Parent class (Exception) already knows **how to store and display error messages**.
- Your custom class just **passes the message up** instead of implementing it again.

```
text

CustomException -> passes "Age invalid" to Exception -> Exception stores it
```

◆ 4. Example

```
java

class InvalidAgeException extends Exception {
    public InvalidAgeException(String msg) {
        super(msg); // pass the message to Exception class
    }
}

public class Test {
    public static void main(String[] args) {
        try {
            throw new InvalidAgeException("Age must be 18 or above!");
        } catch (InvalidAgeException e) {
            System.out.println("Caught: " + e.getMessage());
        }
    }
}
```

```

        }
    }
}
```

Output:

```
makefile
```

```
Caught: Age must be 18 or above!
```

✓ Key takeaway:

- `super(msg)` → tells the **parent class (Exception)**: "Here's the message, please handle/store it."
- Without `super(msg)`, `getMessage()` would return `null`.

If you want, I can **draw a small diagram** showing **how the message flows from your custom exception to the Exception class** visually—it makes it very easy to remember.

Do you want me to do that?



```
import java.util.*;

class InsufficientBalance extends Exception{
    public InsufficientBalance(String msg){
        super(msg);
    }
}

public class Hello{
    public static void main(String args[]){
        int amount=100;
        int balance=50;
        try{
            if(amount>balance) throw new InsufficientBalance("Insufficient amount balance");
        }catch(InsufficientBalance e){
            System.out.println(e.getMessage());
        }
    }
}
```

Perfect! Let's break down your code **line by line** and explain exactly what's happening.

◆ 1. Custom Exception Class

```
java

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

- You created a **custom checked exception** named `InsufficientBalance`.
- It **extends Exception**, so it inherits all functionality of standard exceptions.
- The constructor `InsufficientBalance(String msg)` calls `super(msg)` → this **passes the error message** to the `Exception` class, so you can later retrieve it with `getMessage()`.

◆ 2. Main Class

```
java

public class Hello{
    public static void main(String args[]){
        int amount=100;
        int balance=50;
```

- You define two integers:
 - `amount` → the money the user wants to withdraw
 - `balance` → the money available in the account

◆ 3. Try-Catch Block

```
java

try{
    if(amount>balance) throw new InsufficientBalance("Insufficient amount balance");
}catch(InsufficientBalance e){
    System.out.println(e.getMessage());
}
```

1. try block

- Checks if `amount` is greater than `balance`.
- If true → **throws** your custom exception:

```
java

throw new InsufficientBalance("Insufficient amount balance");
```

- `throw` creates an **exception object** and passes control to the `catch` block.

2. catch block

- Catches the thrown `InsufficientBalance` exception.

- `e.getMessage()` retrieves the message you passed in the constructor:

```
arduino
"Insufficient amount balance"
```

- Prints the message instead of crashing the program.

◆ 4. Program Output

```
nginx
Insufficient amount balance
```

✓ Key points illustrated:

- Custom exceptions let you define **domain-specific errors**.
- `throw` actually triggers the exception.
- `try-catch` handles the exception **gracefully**, so the program continues without crashing.
- `super(msg)` stores the message for retrieval via `getMessage()`.

If you want, I can also show you a **version using throws in the method signature**, so your `main` method could declare it instead of catching it immediately—this is useful for bigger programs.

Do you want me to do that?



where catch not working

Ah! This is an important concept. Catch blocks **won't work in certain situations**. Let me explain clearly with examples.

◆ 1. When Catch Won't Work

(A) Error occurs (not Exception)

- `catch` works for **Exceptions** (`Exception` and its subclasses).
- **Errors** like `OutOfMemoryError`, `StackOverflowError`, `VirtualMachineError` are **not meant to be caught**.
- You *can* catch them technically, but the program may not recover reliably.

```
java
public class CatchErrorException {
    public static void main(String[] args) {
        try {
            int[] arr = new int[Integer.MAX_VALUE]; // OutOfMemoryError
```

```

        } catch (Exception e) {
            System.out.println("Caught Exception!");
        }
    }
}

```

✓ Output: Program will **crash** → catch doesn't handle **Error**.

(B) Exception type mismatch

- Catch block only handles exceptions of **matching type** or **its superclass**.
- If you try to catch a **different exception**, it won't work.

```

java

public class ExceptionMismatch {
    public static void main(String[] args) {
        try {
            int a = 10 / 0; // ArithmeticException
        } catch (NullPointerException e) {
            System.out.println("Caught NullPointerException");
        }
    }
}

```

✓ Output:

```

pgsql

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

```

- **Reason:** Catch block is for **NullPointerException**, not **ArithmaticException**.

Fix: Catch the correct type or a superclass (**Exception**):

```

java

catch (Exception e) { ... }

```

(C) Exception occurs outside the try block

- Only exceptions **inside** `try { }` are caught.

```

java

public class OutsideTry {
    public static void main(String[] args) {
        int a = 10 / 0; // Not in try
        try {
            System.out.println("Hello");
        } catch (ArithmaticException e) {
            System.out.println("Caught!");
        }
    }
}

```

Output:

```
pgsql
Exception in thread "main" java.lang.ArithmetricException: / by zero
```

- **Reason:** Exception happens outside `try`. Catch never sees it.

(D) Exception thrown in a different thread

- Catch blocks in the main thread **won't catch exceptions in another thread**.

```
java
public class ThreadException {
    public static void main(String[] args) {
        Thread t = new Thread(() -> {
            int x = 5 / 0; // ArithmetricException in new thread
        });
        t.start();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {}
    }
}
```

Output:

```
pgsql
Exception in thread "Thread-0" java.lang.ArithmetricException: / by zero
```

- **Reason:** Exception is in a separate thread → main thread catch doesn't work.

(E) Catching only checked exceptions, but runtime exception occurs

- If you declare a catch for a checked exception but a **runtime exception** occurs, it won't be caught unless you catch `Exception` or `RuntimeException`.

```
java
try {
    int[] arr = new int[5];
    System.out.println(arr[10]); // ArrayIndexOutOfBoundsException
} catch (IOException e) {
    System.out.println("Caught IOException");
}
```

Output: Crashes because `ArrayIndexOutOfBoundsException` is not `IOException`.

◆ 2. Summary – When Catch Fails

1. Exception is **outside try block**.

2. Catch type **doesn't match** the thrown exception.

3. Exception is an **Error** (JVM-level).

4. Exception occurs in **another thread**.

If you want, I can make a **small diagram showing "throw → try → catch → finally" flow and where catch fails**, so it's visually easy to understand.

Do you want me to do that?



in my code , no need for catch, because it throw the error

Ah! I understand exactly what you mean. Let's analyze your code in that context.

Your code:

```
java

import java.util.*;

class InsufficientBalance extends Exception{
    public InsufficientBalance(String msg){
        super(msg);
    }
}

public class Hello{
    public static void main(String args[]){
        int amount=100;
        int balance=50;
        try{
            if(amount>balance) throw new InsufficientBalance("Insufficient amount
balance");
        }catch(InsufficientBalance e){
            System.out.println(e.getMessage());
        }
    }
}
```

1 Checked vs Unchecked Exception

- **InsufficientBalance extends Exception**, which is a **checked exception**.
- **Checked exceptions must** be either:
 1. **Caught** using **try-catch**, or
 2. **Declared with throws** in the method signature.

2 Can we remove the catch?

Yes, but then you must **declare that main throws this exception**:

```
java

public class Hello{
    public static void main(String args[]) throws InsufficientBalance {
        int amount=100;
        int balance=50;
        if(amount>balance)
            throw new InsufficientBalance("Insufficient amount balance");
    }
}
```

- Now `main` does **not handle** the exception.
- The exception will **propagate** to the JVM.
- JVM will print the stack trace and terminate the program:

```
php

Exception in thread "main" InsufficientBalance: Insufficient amount balance
at Hello.main(Hello.java:7)
```

-  So, the **catch block is not needed** if you declare `throws InsufficientBalance` and are okay with the program terminating on error.

3 Key Point

- Checked exceptions** → must either **catch** or **declare with throws**.
- Unchecked exceptions (RuntimeException)** → optional to catch, program will throw it automatically if uncaught.

If you want, I can rewrite your code as a **clean version without try-catch**, using `throws`, and also show a **version with runtime exception** so you don't need `throws` at all.

Do you want me to do that?



```
import java.util.*;

class InsufficientBalance extends Exception{
    public InsufficientBalance(String msg){
        super(msg);
    }
}

public class Hello{
    public static void main(String args[]){
        int amount=100;
        int balance=50;
        try{
            if(amount>balance) throw new InsufficientBalance("Insufficient amount balance");
        }
    }
}
```

```

        }catch(InsufficientBalance e){
            System.out.println(e.getMessage());
        }

    }
}"

```

then how to make catch run

Ah! I see exactly what you're asking. You want the **catch block to actually run** in your code. Let's go step by step.

◆ 1. Your current code

```

java

int amount = 100;
int balance = 50;
try {
    if (amount > balance)
        throw new InsufficientBalance("Insufficient amount balance");
} catch (InsufficientBalance e) {
    System.out.println(e.getMessage());
}

```

In this code, the **catch block already runs**, because:

- `amount > balance → 100 > 50 → true`
- `throw new InsufficientBalance(...)` executes → exception is thrown
- Catch block matches `InsufficientBalance` → it **catches it**
- `System.out.println(e.getMessage())` prints the message

Output:

```

nginx

Insufficient amount balance

```

So your catch **is running already**.

◆ 2. Why catch might not run

Catch will **not run** only if:

1. The condition for throw is false

```

java

int amount = 30; // less than balance
int balance = 50;

```

```
if(amount > balance) throw new InsufficientBalance("msg"); // condition false
```

- Here **throw** never executes → catch block never runs.

2. Thrown exception type does not match catch type

```
java

try {
    throw new ArithmeticException("Divide by zero");
} catch(InsufficientBalance e) { // type mismatch
    System.out.println("Caught");
}
```

- Catch **does not run** because **ArithmeticException** ≠ **InsufficientBalance**.

◆ 3. How to ensure catch runs

- Make sure **the exception is actually thrown**:

```
java

int amount = 100;
int balance = 50; // amount > balance, condition true
try {
    if(amount > balance)
        throw new InsufficientBalance("Insufficient amount balance");
} catch(InsufficientBalance e) {
    System.out.println("Catch executed: " + e.getMessage());
}
```

✓ Output:

```
vbnnet

Catch executed: Insufficient amount balance
```

- Or for testing, you can **always throw it**:

```
java

try {
    throw new InsufficientBalance("Test catch block");
} catch(InsufficientBalance e) {
    System.out.println("Catch executed: " + e.getMessage());
}
```

◆ Summary

- Catch **only runs if**:
 1. An exception **is thrown inside try**
 2. The exception **matches the type** in catch

- If your `amount <= balance`, or you throw a different type, catch won't execute.

If you want, I can **rewrite your code** so it's a **full working example with both catch running and finally block**, which is best practice for exceptions.

Do you want me to do that?



Throwable (diff & similarities)

Sure! Let's go **deep into Throwable in Java**, its differences and similarities with **Exception** and **Error**, with examples and explanation.

◆ 1. What is Throwable?

- **Definition:**

`Throwable` is the **superclass of all errors and exceptions** in Java.

- Any object that can be **thrown using throw** must be an instance of `Throwable` or its subclass.
- Package: `java.lang`

Hierarchy:

```
php
java.lang.Object
    └── java.lang.Throwable
        ├── java.lang.Error
        └── java.lang.Exception
```

◆ 2. Key Points About Throwable

1. Can be **thrown using throw**.
2. Can be **caught using catch**.
3. Provides **methods** like:
 - `getMessage()` → returns the error message.
 - `printStackTrace()` → prints the exception/error stack.
 - `toString()` → returns a short description.
4. Parent class of **both Exceptions and Errors**.

◆ 3. Differences Between Throwable, Exception, and Error

Feature	Throwable	Exception	Error
Superclass of	All Errors & Exceptions	All checked/unchecked exceptions	Serious JVM/runtime errors
Recoverable?	Depends on subclass	Usually yes (recoverable)	Usually no (not recoverable)
Checked or Unchecked	N/A	Checked (except RuntimeException)	Unchecked
Must be caught?	Only checked exceptions must be caught	Checked exceptions must be caught	Not usually caught
Examples	N/A (abstract parent)	IOException, NullPointerException	OutOfMemoryError, StackOverflowError
Can be thrown?	Yes	Yes	Yes
Use	Base class for all throwable objects	Application level error handling	JVM/system level error handling

◆ 4. Similarities Between Throwable, Exception, and Error

1. All **inherit from Throwable**.
2. All can be **thrown using throw**.
3. All can be **caught using catch** (though catching Errors is not recommended).
4. All have **methods: getMessage(), toString(), printStackTrace()**.

◆ 5. Example: Using Throwable

```
java

public class ThrowableExample {
    public static void main(String[] args) {
        try {
            throw new Throwable("This is a Throwable object");
        } catch (Throwable t) {
            System.out.println("Caught: " + t.getMessage());
            t.printStackTrace();
        }
    }
}
```

Output:

```
php

Caught: This is a Throwable object
java.lang.Throwable: This is a Throwable object
    at ThrowableExample.main(ThrowableExample.java:4)
```

◆ 6. Key Takeaways

- **Throwable** is the **root of all exceptions and errors**.

- **Exception** → recoverable, used in applications.
- **Error** → serious system-level issues, usually not handled.
- Both Exception and Error **inherit common methods** from Throwable.

If you want, I can make a **diagram showing Throwable → Exception → RuntimeException and Throwable → Error hierarchy with checked/unchecked info**, which makes it super easy to memorize.

Do you want me to do that?



what to do if error occurs , instead of exception

Ah! This is a very important distinction. Let's carefully go through **what to do if an Error occurs** in Java instead of a regular Exception.

◆ 1. Recap: Exception vs Error

Feature	Exception	Error
Recoverable?	Usually yes	Usually no (serious system issues)
Checked/Unchecked	Checked (must handle) or Unchecked	Unchecked (system/JVM errors)
Handling required?	Yes	Rarely, only in special cases
Example	IOException, NullPointerException	OutOfMemoryError, StackOverflowError

Key: Errors are **usually not meant to be handled** because they indicate **serious problems in JVM or system**.

◆ 2. Can We Catch Errors?

Technically, **yes**, because **Error** extends **Throwable**. But:

- Catching **Error** is **not recommended** unless you have a very specific reason.
- Program may be in an **unstable state**, and recovery may fail.

```
java

public class ErrorHandlingDemo {
    public static void main(String[] args) {
        try {
            // This may cause StackOverflowError
            recursive();
        } catch (StackOverflowError e) {
            System.out.println("Caught Error: " + e);
        }
    }

    static void recursive() {
        recursive(); // infinite recursion
    }
}
```

```
}
```

Output:

```
javascript
Caught Error: java.lang.StackOverflowError
```

- The catch works, but the program may still behave unexpectedly.

◆ 3. Proper Strategy When an Error Occurs

1. Prevent the error

- Most errors occur due to **resource misuse**. For example:
 - **OutOfMemoryError** → too many objects in memory → optimize code.
 - **StackOverflowError** → infinite recursion → fix the recursion.

2. Log and terminate gracefully

- Catching the error to **log it** is okay, then **exit** safely:

```
java

try {
    // risky operation
} catch (OutOfMemoryError e) {
    System.out.println("System ran out of memory. Logging and terminating.");
    e.printStackTrace();
    System.exit(1);
}
```

3. Use **finally** block for cleanup

- Even if an Error occurs, **finally** block **still runs** for resource cleanup:

```
java

try {
    // code that may cause error
} finally {
    System.out.println("Cleaning up resources...");
}
```

4. Avoid catching errors blindly

- Don't use **catch(Throwable t)** just to ignore errors — it hides serious problems.

◆ 4. Real-World Analogy

- **Exception** → like a **traffic jam**: you can take a detour and continue your journey.
- **Error** → like a **bridge collapse**: the path is gone, you need to stop and fix the infrastructure first.

◆ 5. Key Takeaways

- Errors are **serious system/JVM-level problems**.
 - Best approach: **prevent, log, cleanup, terminate gracefully**.
 - **Exceptions** → handle with try-catch.
 - **Errors** → usually **don't recover**, just log and exit.
-

If you want, I can make a **flow diagram showing how Exception vs Error is handled in Java** including **try-catch-finally** and JVM termination — this makes it crystal clear visually.

Do you want me to do that?