

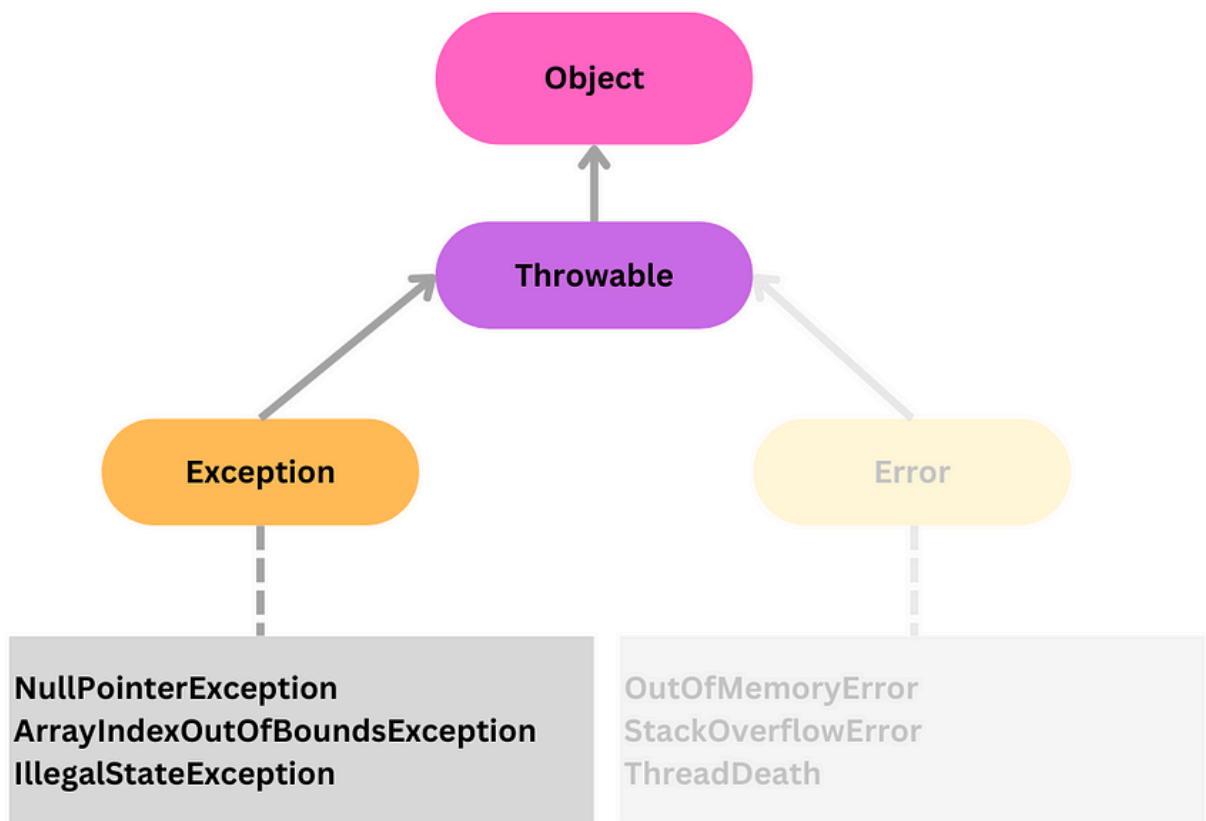
# Exception Handling

## What is an Exception?

In programming, *an exception is an unexpected event or situation that happens while a program is running*, interrupting its usual operation and occasionally leading to crashing the program. If a piece of code faces an unusual situation, it can throw an exception to indicate that something unexpected has occurred. This exception must then be caught and managed by specific code to prevent the program from crashing or behaving unpredictably.

In Java, exceptions are a subclass of the `Throwable` class, which has two main subclasses `Exception` and `Error`.

Exceptions, extending the `Exception` class, are *for conditions that applications can reasonably catch and handle*. On the other hand, errors, extending the `Error` class, *indicate critical issues typically beyond the application's control*. In most of the cases, you don't want to handle Errors in your application, only Exceptions.



## **Common scenarios where exceptions happen.**

1. Trying to divide a number by zero

2. Attempting to access an array element with an index that falls outside the valid range.
3. Trying to access a file that doesn't exist.

As an example let's try to divide a number by zero like below.

```
public static void main(String[] args) {  
  
    System.out.println(10/0);  
  
}
```

When you run the program, it crashes and displays an exception like the following.

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

## Handling Exception

Exception handling includes using constructs like `try`, `catch` and `finally` blocks. The code that you suspect might cause an exception is enclosed within a `try` block.

If an exception happens, it is caught by a corresponding `catch` block that defines the type of exception to manage.

`finally` blocks are used to specify code that should run whether an exception occurred or not, Using a `finally` block is optional, but using **try-catch** blocks is mandatory in exception handling.

```
try {
```

```
    // Section where you include the code that might generate  
    exceptions.
```

```
} catch (Exception e) {
```

```
    // Section where you catch the type of exception and
```

```
    // perform necessary actions.
```

```
} finally {
```

```
    // Section where you write a specific block of code that executes
```

```
    // regardless of whether an exception is thrown or not.
```

```
}
```

Now, let's handle the above exception using a **try-catch-finally** block.

```
try {  
  
    System.out.println(10/0);  
  
} catch (Exception e) {  
  
    System.out.println("Number cannot be divide by zero");  
  
} finally {  
  
    System.out.println("finished run");  
  
}
```

```
Number cannot be divide by zero  
finished run
```

Notice that the program no longer crashes as it did before. It triggers the `catch` block, executes the code inside it, and then

proceeds to the code within the `finally` block. The `finally` block is executed regardless of whether the program encounters an exception or not.

Let's try to divide the number 10 by 2 instead of 0 and run the program.

```
5  
finished run
```

This operation does not result in an exception, and the program runs smoothly as expected. And the `finally` block of code is executed this time as well.

## Multiple Catch blocks

In Java, multiple catch blocks offer a more granular way to handle various exception scenarios than a single try-catch structure. If a section of code might generate different types of



exceptions, developers can use multiple `catch` blocks to handle each type of exception.

Each `catch` block is created to handle a particular exception, and they are arranged from the *most specific to the most general*. This ensures precise handling of different exceptional conditions. Each catch blocks are assessed one after the other, and only the first one that matches the type of the thrown exception is executed.

Let's take a look at the example code. The code is designed to simulate two exceptions.

```
public static void main(String[] args) {  
  
    try {  
  
        // Simulate a NullPointerException
```

```
String text = null;
```

```
int length = text.length();
```

```
// Simulate an ArithmeticException
```

```
int result = 5 / 0;
```

```
} catch (NullPointerException e) {
```

```
    System.out.println("Caught NullPointerException: " +  
e.getMessage());
```

```
} catch (ArithmeticException e) {
```

```
    System.out.println("Caught ArithmeticException: " +  
e.getMessage());
```

```
    } catch (Exception e) {  
  
        System.out.println("Caught generic Exception: " +  
e.getMessage());  
  
    } finally {  
  
        System.out.println("Finally block always executes.");  
  
    }  
  
}
```

If you run this code as it is, the `NullPointerException` will be thrown first, and the corresponding catch block will be executed without proceeding to the other exception.

```
dceeee08ae026466e75021faf3c/redhat.java/jdt_ws/exception_a1e5725a/bin Example  
Caught NullPointerException: Cannot invoke "String.length()" because "text" is null  
Finally block always executes.
```

And if you attempt to simulate an `ArithmeticException` from the multiple catch blocks, the corresponding catch block output will be generated.

```
public static void main(String[] args) {  
  
    try {  
  
        String text = "Hello World";  
  
        int length = text.length();  
  
  
        // Simulate an ArithmeticException  
  
        int result = 5 / 0;  
    }  
}
```

```
    } catch (NullPointerException e) {
```

```
        System.out.println("Caught NullPointerException: " +  
e.getMessage());
```

```
    } catch (ArithmeticException e) {
```

```
        System.out.println("Caught ArithmeticException: " +  
e.getMessage());
```

```
    } catch (Exception e) {
```

```
        System.out.println("Caught generic Exception: " +  
e.getMessage());
```

```
    } finally {
```

```
        System.out.println("Finally block always executes.");
```

```
}
```

```
}
```

```
dceeee08ae026466e75021faf3c/redhat.java/jdt_ws/exception_a1e5725a/bin Example  
Caught ArithmeticException: / by zero  
Finally block always executes.
```

After handling specific exceptions, there is often a `catch` block to handle more general exceptions.

Also, If you want to execute a common code block for multiple exceptions, you can combine the exceptions using `|` and create a single `catch` block for them, like the below example.

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

```
    try {
```

```
        String text = "Hello World";
```

```
        int length = text.length();
```

```
// Simulate an ArithmeticException
```

```
int result = 5 / 0;
```

```
    } catch (NullPointerException | ArithmeticException e) {
```

```
        System.out.println("Caught Specific Exception: " +  
e.getMessage());
```

```
    } catch (Exception e) {
```

```
        System.out.println("Caught generic Exception: " +  
e.getMessage());
```

```
    } finally {
```

```
        System.out.println("Finally block always executes.");  
  
    }  
  
}
```

## Finally Block

Even though using `finally` block is Optional. When you use it you should be very careful because `finally` block *runs always, even if you've used a `return` statement in the above try-catch block.*

Look at the example below.

In the typical programming flow, any code below a `return` statement within a block won't execute. So, you might assume that the below function `getANumber()` should return 10 and stop.

```
public static void main(String[] args) {  
  
    System.out.println("Number i got is "+getANumber());  
}
```



```
}
```

```
private static int getANumber() {
```

```
    try {
```

```
        return 10;
```

```
    } catch (Exception e) {
```

```
        return 5;
```

```
    } finally {
```

```
        return -100;
```

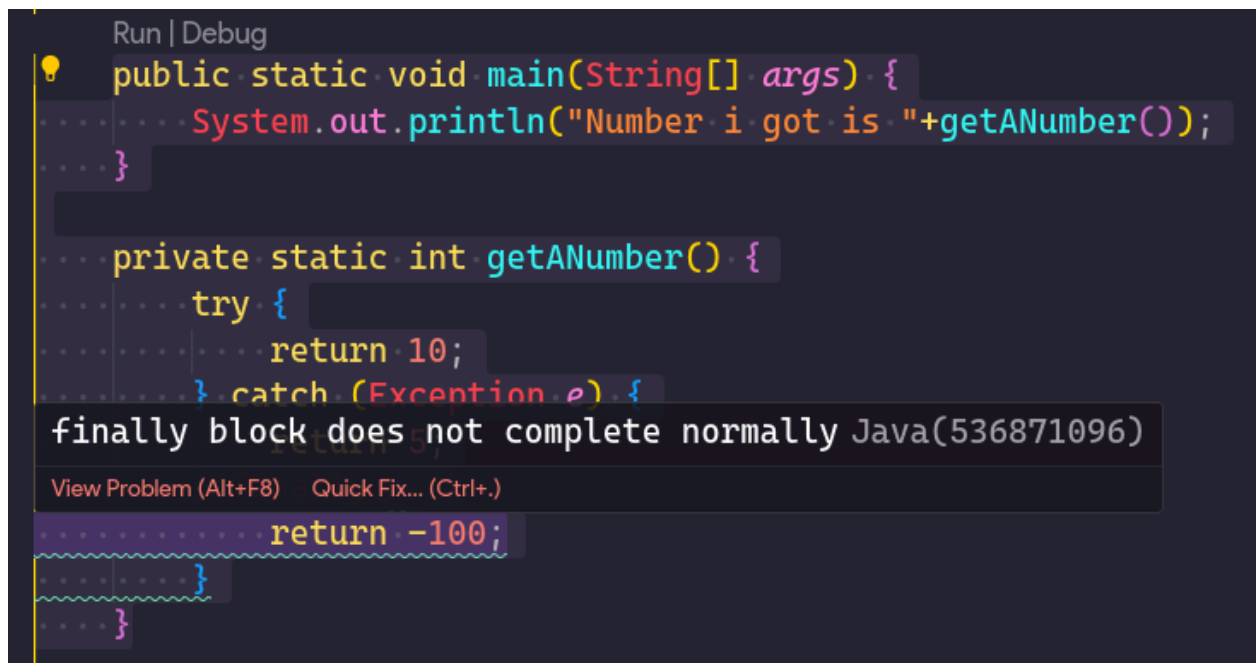
```
    }
```

```
}
```

However, when we run it we get the following output.

```
dceeee08ae026466e75021faf3c/redhat.java/jdt_ws/exception_a1e5725a/bin Example  
Number i got is -100
```

This means because of `finally` block *always runs no matter what*, The `return` statement of `finally` block is overriding the `return` statement of `try` or `catch` block. This can cause crazy unexpected behaviours. So in general try to avoid using `return` statement in `finally` block. As you can see below, VS-Code is warning us about that too,



```
Run | Debug  
public static void main(String[] args) {  
    System.out.println("Number i got is "+getANumber());  
}  
  
private static int getANumber() {  
    try {  
        return 10;  
    } catch (Exception e) {  
        finally block does not complete normally Java(536871096)  
        return -100;  
    }  
}
```

The screenshot shows a Java IDE with a code editor. The code defines a `main` method that calls `getANumber()` and prints the result. The `getANumber()` method has a `try-catch` block. Inside the `try` block, it returns 10. The `catch` block is empty. Below the `catch` block, there is a `finally` block that contains `return -100;`. A red squiggly line underlines the `return -100;` statement, and a tooltip message says "finally block does not complete normally Java(536871096)". Below the tooltip, there are two buttons: "View Problem (Alt+F8)" and "Quick Fix... (Ctrl+.)".

## Best Practices:

1. Handle only specific exceptions: Catching and handling all exceptions is not a good practice. Instead, try to catch only specific exceptions that you are expecting. This helps to improve the readability and maintainability of the code.
2. Use meaningful exception messages: When you catch an exception, always use meaningful and informative messages. This helps to identify the root cause of the exception quickly and take appropriate actions.
3. Log exceptions: Logging exceptions is essential for debugging and troubleshooting. Use a logging framework like Log4j or Java Util Logging to log exceptions.
4. Don't swallow exceptions: Swallowing exceptions means catching an exception but not doing anything about it. This is a bad practice as it can lead to unexpected behavior and errors in the application.

5. **Throw checked exceptions:** Checked exceptions are those that the Java compiler requires you to catch or declare in the method signature. Always throw checked exceptions instead of catching them silently.
6. **Use finally block for resource cleanup:** The finally block is always executed, whether an exception occurs or not. Use the finally block to release any resources that were acquired in the try block, such as closing database connections, files, or streams.
7. **Use exception chaining:** Exception chaining is a technique where you wrap an exception in another exception. This helps to preserve the original exception's stack trace and provide more meaningful information about the error.
8. **Use custom exceptions:** Custom exceptions are user-defined exceptions that are specific to an application. Use custom exceptions to provide more meaningful and descriptive errors to the users.

9. Don't catch Error: Errors are severe exceptions that cannot be recovered from. Avoid catching Error in your code as it can lead to unpredictable behavior.
10. Keep exception handling code separate: Keep the exception handling code separate from the main logic of the application. This makes the code more readable and easier to maintain.