

# Exceptions

# Exceptions

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

When an exception occurs within a class or method, the method/class creates an exception object and hands the results to the runtime system (JVM).

What are some possible exceptions we can have in our code?




# Exception Handling

Exception Handling in Java is one of the effective means to handle the runtime errors so that the regular flow of the application can be preserved.

Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.



# Some built-in exceptions in Java

- Exception: This is the parent class of other exceptions
  - IndexOutOfBoundsException
  - IllegalArgumentException
  - IllegalStateException
  - StringIndexOutOfBoundsException
  - ArithmeticException
  - ClassNotFoundException
  - FileNotFoundException
  - IOException
  - SQLException
  - NullPointerException
  - NoSuchMethodException
  - NumberFormatException
- 


# How can Java catch exceptions?

## Try and Catch Statements

- a. **Try:** Defines a block of code to be tested for error when it runs.
- b. **Catch:** t defines a block of code that is executed if there is an error in the try block.

You can use the **try/catch** block to handle exceptions:

```
try{  
    // block of code that could throw an exception.  
}catch(ExceptionType variable){  
    // block of code executed when there is an ExceptionType  
}
```




# Try/Catch Statements

You may chain multiple catch statement if your code throws more than one exception:


```
try{  
    // block of code that could throw an exception.  
}catch(ExceptionType variable){  
    // block of code executed when there is an ExceptionType  
}catch(AnotherExceptionType variable){  
    // block of code executed when there is an AnotherExceptionType  
}
```

Use **try/catch** to print messages that any user can understand (not only programmers)



# try/catch/finally


```
try{  
    // block of code that could throw an exception.  
}catch(ExceptionType variable){  
    // block of code executed when there is an ExceptionType  
}finally{  
    // It is optional. The finally block always gets executed,  
    // whether an exception occurred or not.  
}
```



# Example

Let's write a Java program where you will have three integer variables ( $a=10$ ,  $b=0$ , and  $c=a/b$ ) and see what happens.

```
try{  
    int a = 0;  
    int b = 0;  
    int c = a/b;  
}catch(ArithmeticException e){  
    System.out.println("Error division by zero");  
}
```





# Exceptions Advantages

Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code. For example, consider the pseudocode method here that reads an entire file into memory.

```
readFile {  
    open the file;  
    determine its size;  
    allocate that much memory;  
    read the file into memory;  
    close the file;  
}
```

\*\* See file



# Cause an exception - Throw new exception

So far we have been printing errors when something in our code could cause an issue.

Instead of printing, we can create custom exceptions using the keyword **throw** in Java.

You then specify the Exception object you wish to throw.

Every Exception includes a message which is a human-readable error description.

```
throw new Exception("Your custom message here");
```




# Keyword throws

**Throws** is a keyword used to indicate that a method could throw this type of exception. The caller has to handle the exception using a try-catch block or propagate the exception. We can throw either checked or unchecked exceptions.


The **throws** keyword allows the compiler to help you write code that handles this type of error:

```
void testMethod() throws ArithmeticException, ArrayIndexOutOfBoundsException {  
  
}
```



# Print the stack trace

```
try{  
    CheckAge c = new CheckAge();  
    c.check(15);  
}catch(Exception e){  
    System.out.println("Exception was found.");  
    e.printStackTrace();  
}
```



# Summary

1. **throw** – We know that if an error occurs, an exception object is getting created and then Java runtime starts processing to handle them. Sometimes we might want to generate exceptions explicitly in our code. The **throw** keyword is used to throw exceptions to the runtime to handle it.
  2. **throws** – When we are throwing an exception in a method and not handling it, then we have to use the **throws** keyword in the method signature to let the caller program know the exceptions that might be thrown by the method. The caller method might handle these exceptions or propagate them to its caller method using the **throws** keyword. We can provide multiple exceptions in the **throws** clause.
  3. **try-catch** – We use the **try-catch** block for exception handling in our code. **try** is the start of the block and **catch** is at the end of the **try** block to handle the exceptions. We can have multiple **catch** blocks with a **try** block. The **try-catch** block can be nested too. The **catch** block requires a parameter that should be of type **Exception**.
  4. **finally** – the **finally** block is optional and can be used only with a **try-catch** block. Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use the **finally** block. The **finally** block always gets executed, whether an exception occurred or not.
- 