

Course Code: SL3001	Course: Software Development and construction
Instructor(s):	Miss Nida Munawar, Miss Abeeha Sattar

Lab # 05

Exception Handling

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.

Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

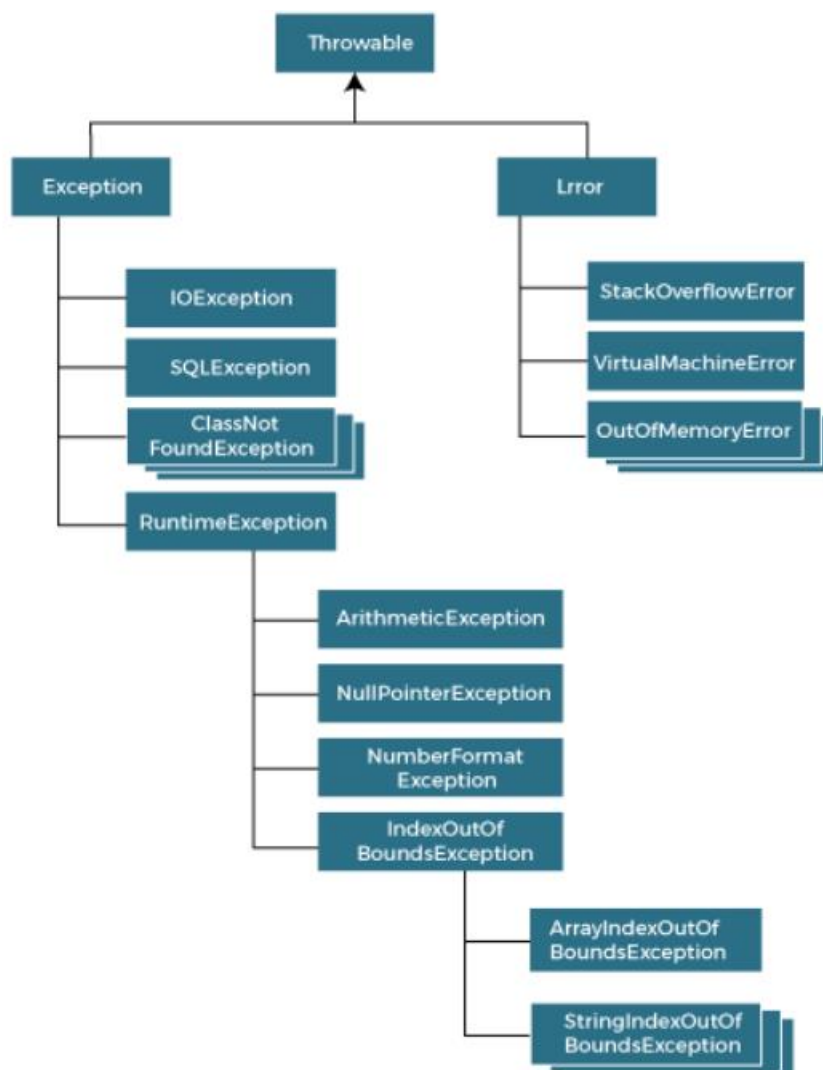
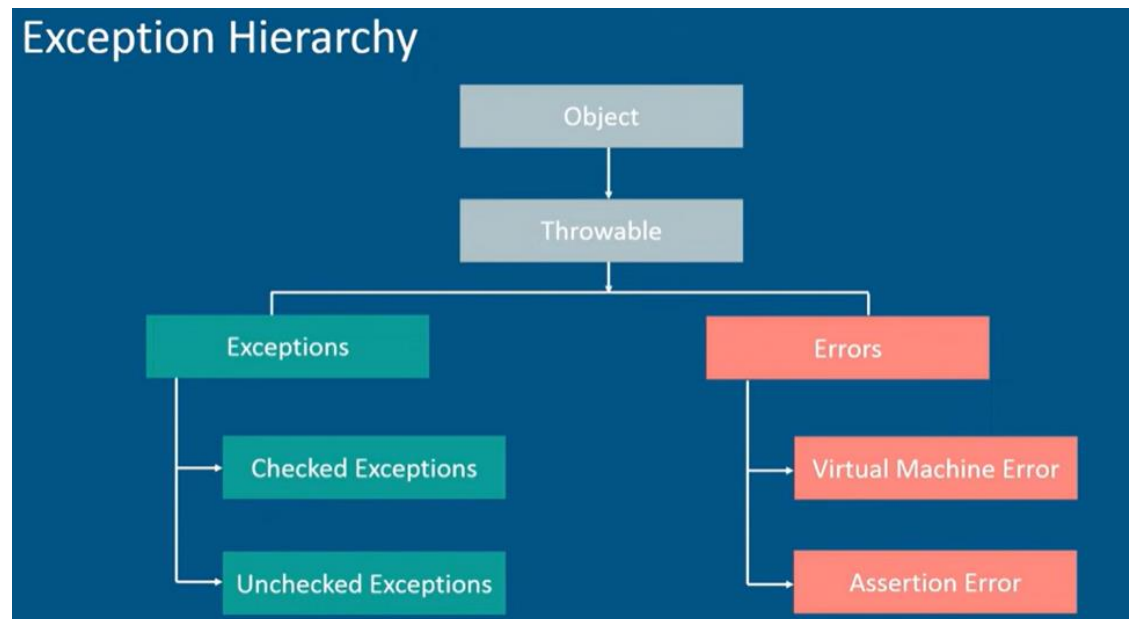
1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5; *//exception occurs*
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed.

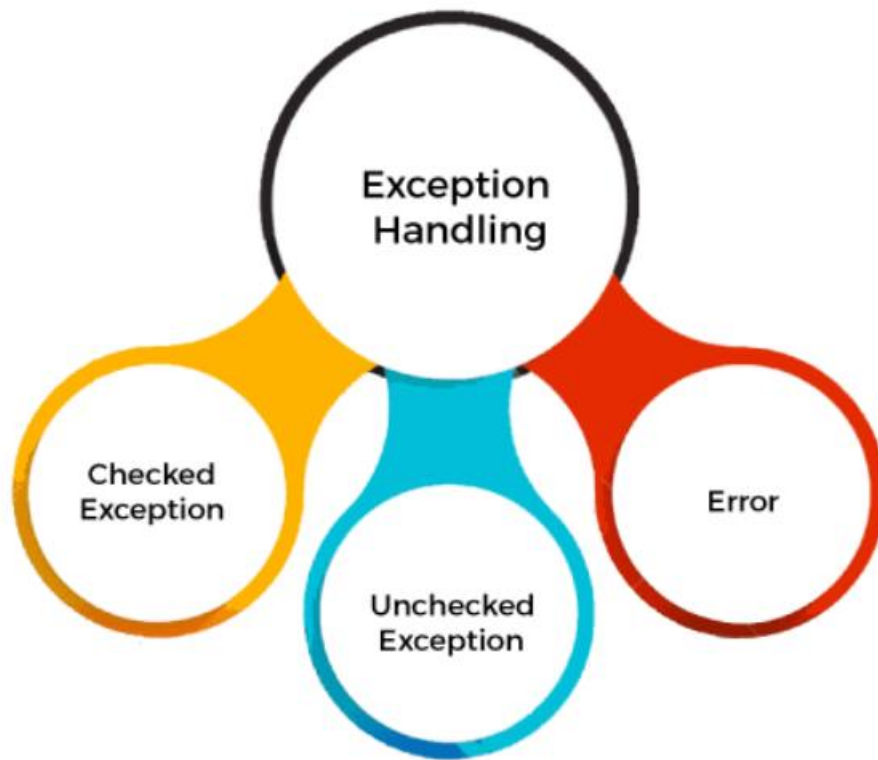
However, when we perform exception handling, the rest of the statements will be executed.

That is why we use exception handling in [Java](#)

Hierarchy of Java Exception classes



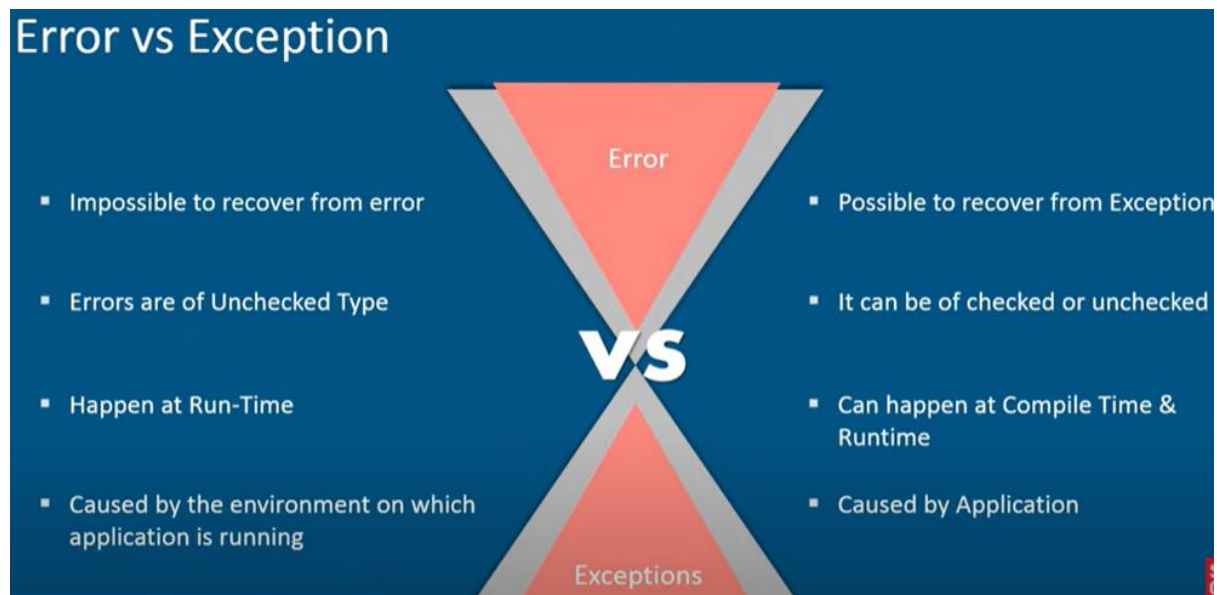
Types of Java Exceptions



There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

Error vs Exceptions



Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

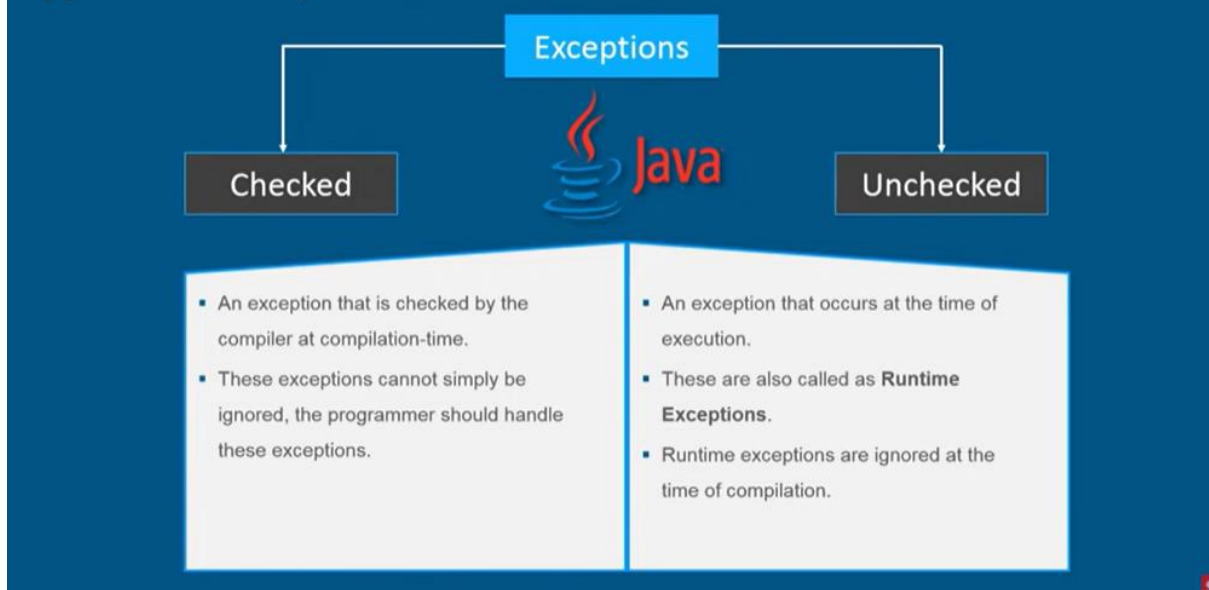
2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

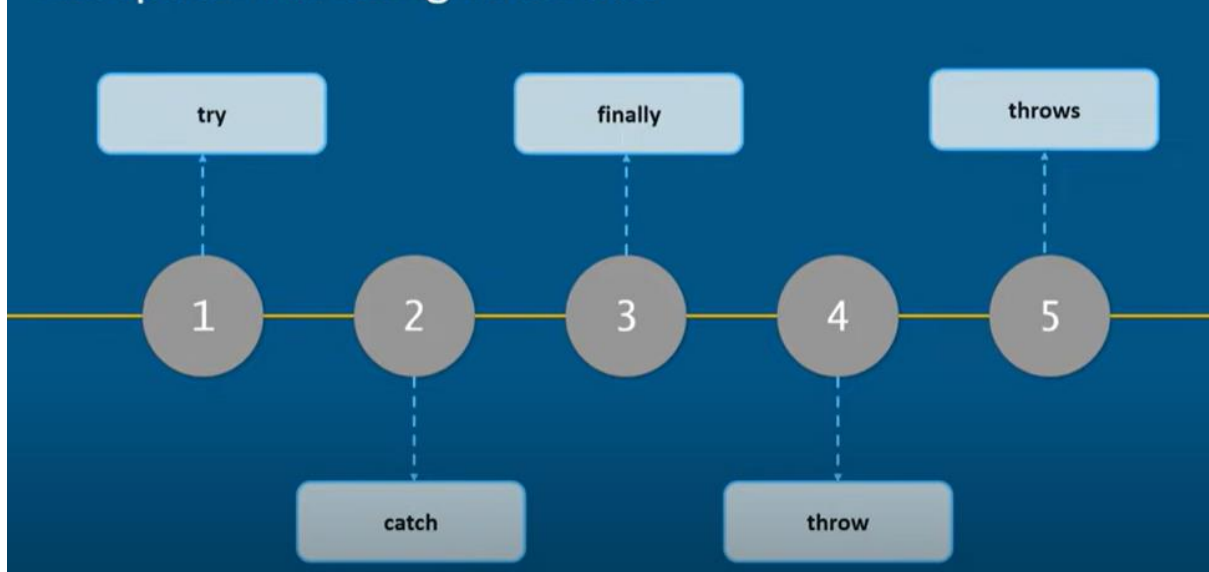
Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Types of Exceptions



Java Exception Keywords

Exception Handling Methods



Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch) and handle it in some rational manner. Systemgenerated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature. Any code that absolutely must be executed after a try block completes is put in a finally block.

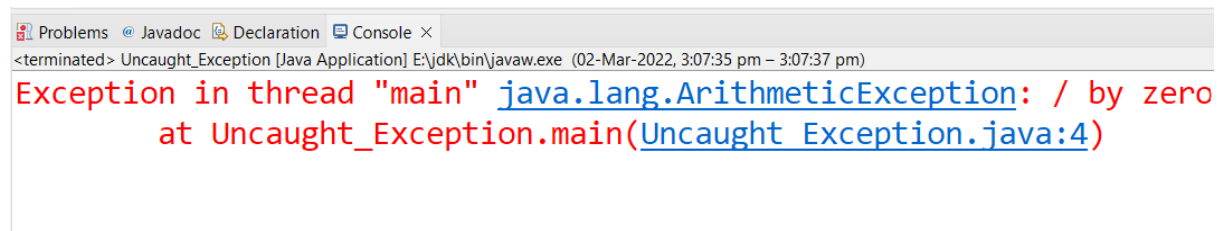
Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Uncaught Exceptions

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them

```
public class Uncaught_Exception {
    public static void main(String[] args) {
        int d = 0 ;
        int a = 2/d;
        System.out.println("hello nida");
    }
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception.



The screenshot shows an IDE's console window with the following text:

```
<terminated> Uncaught_Exception [Java Application] E:\jdk\bin\javaw.exe (02-Mar-2022, 3:07:35 pm - 3:07:37 pm)
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Uncaught_Exception.main(Uncaught_Exception.java:4)
```

Notice how the **class name**, Uncaught_Exception; the **method name**, main; the **filename**, Uncaught_Exception.java; and the **line number**, 4, are all included in the simple stack trace. Also, notice that the **type of exception** thrown is a subclass of Exception called ArithmeticException, which more specifically describes what type of error happened.

Basic Example format of Exception

```
class Exception{
    public static void main(String args[]){
        try{
            //code that may raise exception
        }
        catch(Exception e){
            // rest of the program
        }
    }
}
```

Exception Handling Methods

try

catch

finally

throw

throws

Used to specify a block where we should place exception code.

Syntax:

```
try{
    //code that throws exception
}catch(Exception_class_Name){}
```

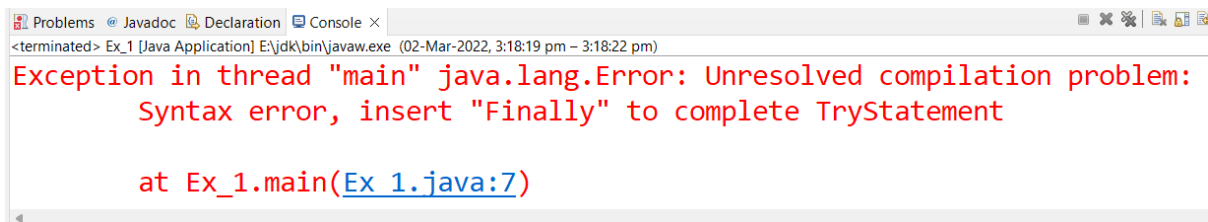
Try without catch

```
public class Ex_1 {
    public static void main(String[] args) {
        int i = 0;
```

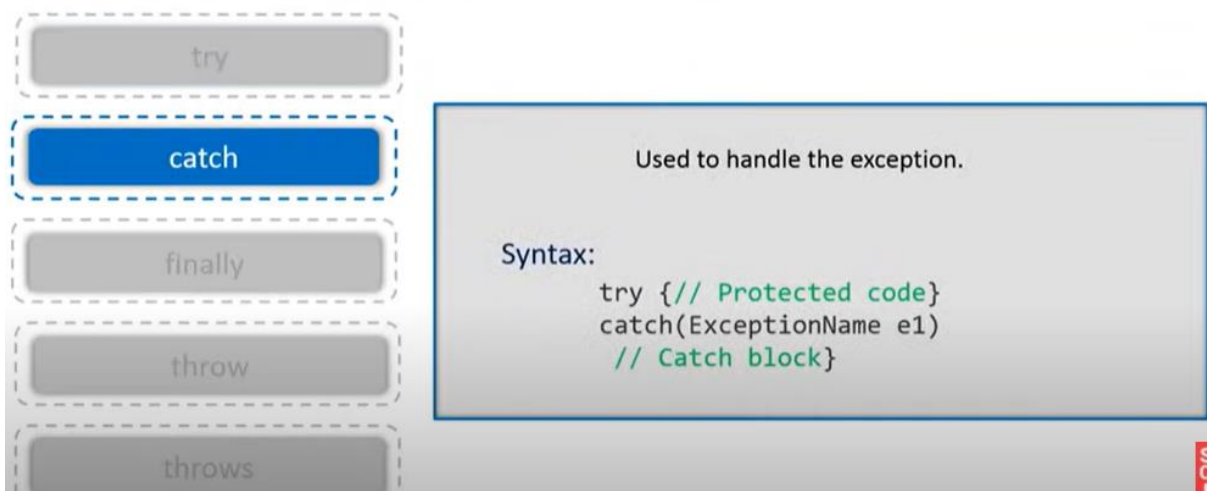
```

    int j = 0;
    try {
        int k = i/j;//critical statement
inside catch
    }
    }

```



Exception Handling Methods



Try with catch

```

public class Ex_1 {
public static void main(String[] args) {
    int i = 0;
    int j = 0;
    try {
        int k = i/j;
    }
    catch(ArithmeticException e) {

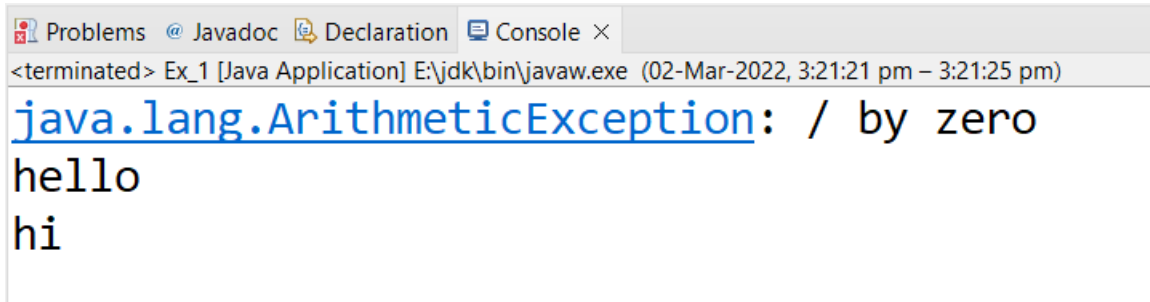
```



```

        System.out.println(e);
    }
    System.out.println("hello");
    System.out.println("hi");
}

```



```

Problems @ Javadoc Declaration Console ×
<terminated> Ex_1 [Java Application] E:\jdk\bin\javaw.exe (02-Mar-2022, 3:21:21 pm – 3:21:25 pm)
java.lang.ArithmeticException: / by zero
hello
hi

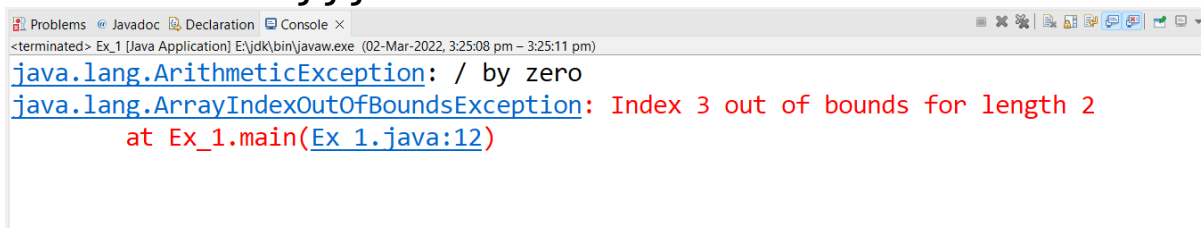
```

Multiple Try and catch with printStackTrace method

```

public class Ex_1 {
    public static void main(String[] args) {
        int i = 0;
        int j = 0;
        try {
            int k = i/j;
        }
        catch(ArithmeticException e) {
            System.out.println(e);}
        try {
            int[] n = new int[2];
            n[3]=4;
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```



```

Problems @ Javadoc Declaration Console ×
<terminated> Ex_1 [Java Application] E:\jdk\bin\javaw.exe (02-Mar-2022, 3:25:08 pm – 3:25:11 pm)
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 2
    at Ex_1.main(Ex 1.java:12)

```

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where `ArithmeticException` occurs If we divide any number by zero, there occurs an `ArithmeticException`.

1. `int a=50/0;//ArithmeticException`

2) A scenario where `NullPointerException` occurs if we have a null value in any variable , performing any operation on the variable throws a `NullPointerException`.

1. `String s=null;`
2. `System.out.println(s.length());//NullPointerException`

3) A scenario where `NumberFormatException` occur if the formatting of any variable or number is mismatched, it may result into `NumberFormatException`. Suppose we have a string variable that has characters; converting this variable into digit will cause `NumberFormatException`

1. `String s="abc";`
2. `int i=Integer.parseInt(s);//NumberFormatException`

4) A scenario where `ArrayIndexOutOfBoundsException` occurs when an array exceeds to it's size, the `ArrayIndexOutOfBoundsException` occurs. there may be other reasons to occur `ArrayIndexOutOfBoundsException`. Consider the following statements

1. `int a[]=new int[5];`
2. `a[10]=50; //ArrayIndexOutOfBoundsException`

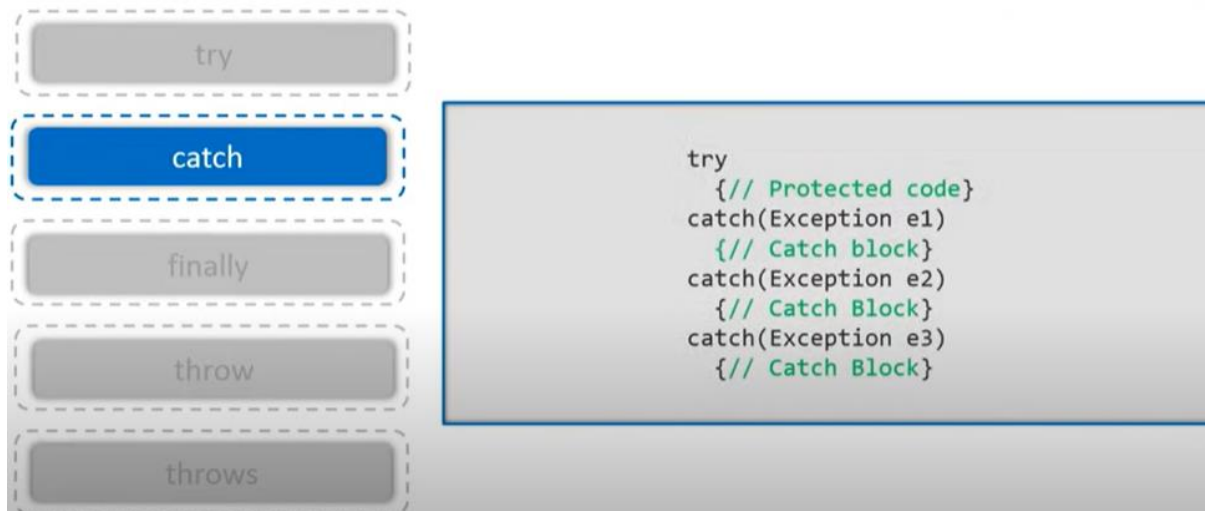
Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for `ArithmeticException` must come before catch for `Exception`.

Exception Handling Methods

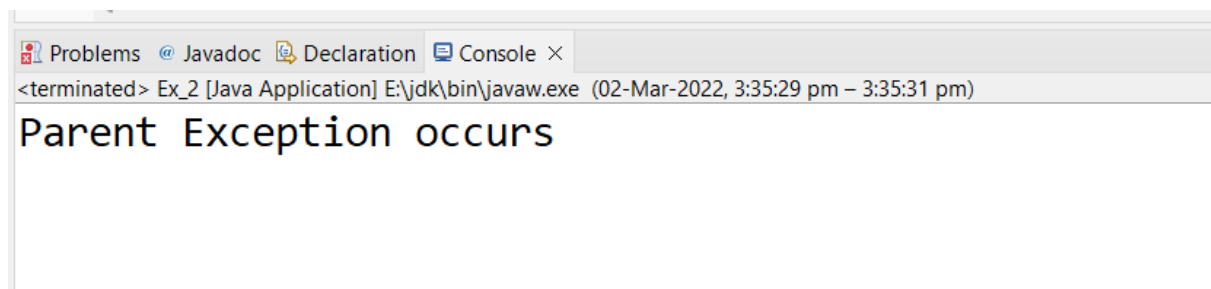


```
public class MultipleCatchBlock1 {
    public static void main(String[] args) {
        try{
            int a[]=new int[5];
            a[5]=30/0;
        }
        catch(ArithmeticException e)
        {
            System.out.println("Arithmetic Exception occurs");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("ArrayIndexOutOfBoundsException Exception occurs");
        }
        catch(Exception e)
        {
            System.out.println("Parent Exception occurs");
        }
        System.out.println("rest of the code");
    }
}
```

```
Arithmetic Exception occurs
rest of the code
```

In this example, we generate `NullPointerException`, but didn't provide the corresponding exception type. In such case, the catch block containing the parent exception class `Exception` will invoked.

```
public class Ex_2 {  
    public static void main(String[] args) {  
        try{  
            String s=null;  
            System.out.println(s.length());  
        }  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
  
            System.out.println("ArrayIndexOutOfBoundsException  
Exception occurs");  
        }  
        catch(Exception e)  
        {  
            System.out.println("Parent Exception  
occurs");  
        }  
    }  
}
```

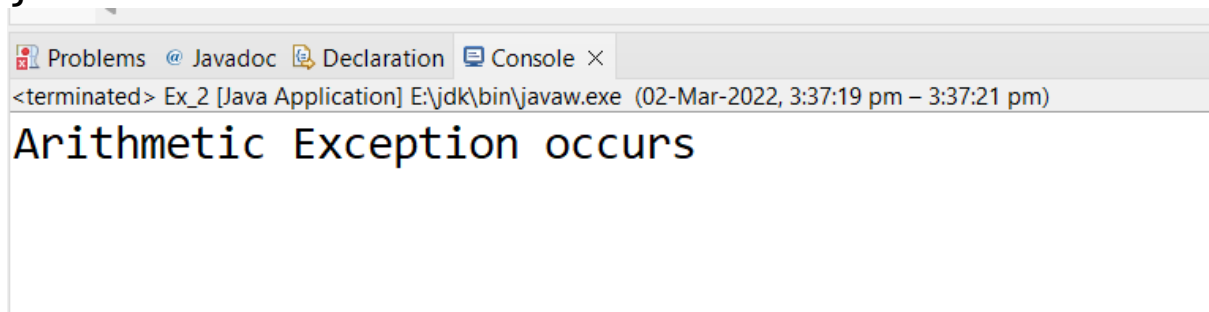


In this example, try block contains two exceptions. But at a time only one exception occurs and its corresponding catch block is executed.

```

public class Ex_2 {
    public static void main(String[] args) {
        try{
            int a[]=new int[5];
            a[5]=30/0;
            System.out.println(a[10]);
        }
        catch(ArithmeticException e)
        {
            System.out.println("Arithmetic
Exception occurs");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("ArrayIndexOutOfBoundsException
Exception occurs");
        }
    }
}

```



an example, to handle the exception without maintaining the order of exceptions (i.e. from most specific to most general).

```

public class Ex_2 {
    public static void main(String[] args) {
        try{
            int a[]=new int[5];
            a[5]=30/0;
        }
    }
}

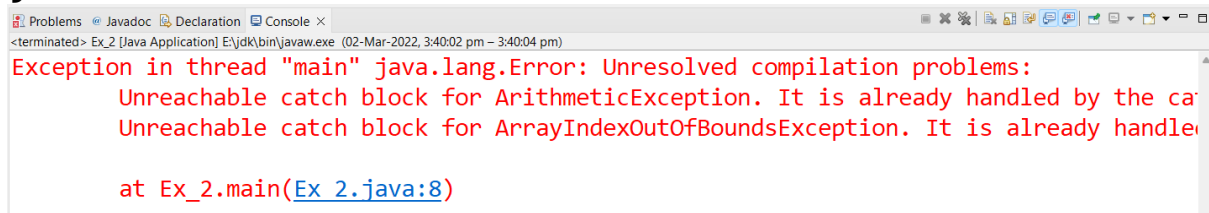
```

```

        catch(Exception
e){System.out.println("common task
completed");}
        catch(ArithmeticException
e){System.out.println("task1 is completed");}

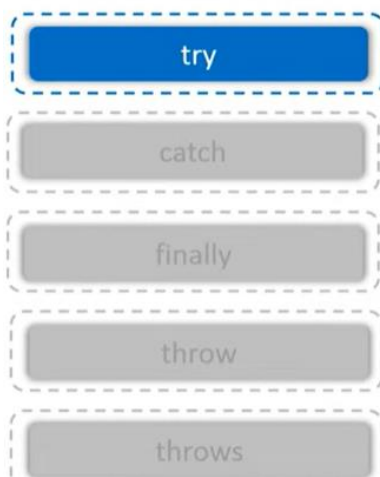
catch(ArrayIndexOutOfBoundsException
e){System.out.println("task 2 completed");}
    System.out.println("rest of the
code...");
}
}

```



Java Nested try block

Exception Handling Methods



```

try{
    try{
        //statements to execute
    }catch(ArithmeticException e)
        {System.out.println(e);}
    try{
        int a[]=new int[5]; a[5]=4;
    }
}

```

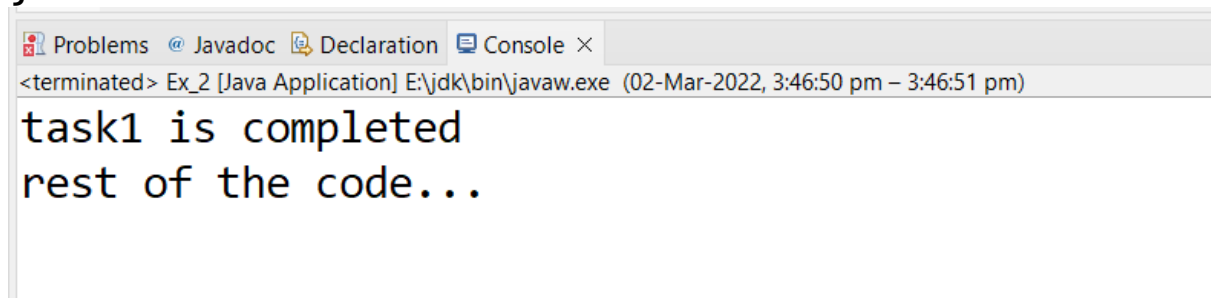
Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

```

public class Ex_2 {
    public static void main(String[] args) {
        try{
            try {
                int a;
                a=30/0;
            }
            catch(ArithmeticException
e){System.out.println("task1 is completed");}
        }
        catch(Exception
e){System.out.println("common task
completed");}
        System.out.println("rest of the
code...");
    }
}

```



The screenshot shows a Java IDE window with a console tab. The console output displays the results of the program execution: "task1 is completed" followed by "rest of the code...". The window title bar indicates the application is "Ex_2 [Java Application]" and the command used is "E:\jdk\bin\javaw.exe". The timestamp shows the execution occurred on 02-Mar-2022 at 3:46:50 pm to 3:46:51 pm.

```

<terminated> Ex_2 [Java Application] E:\jdk\bin\javaw.exe (02-Mar-2022, 3:46:50 pm – 3:46:51 pm)
task1 is completed
rest of the code...

```

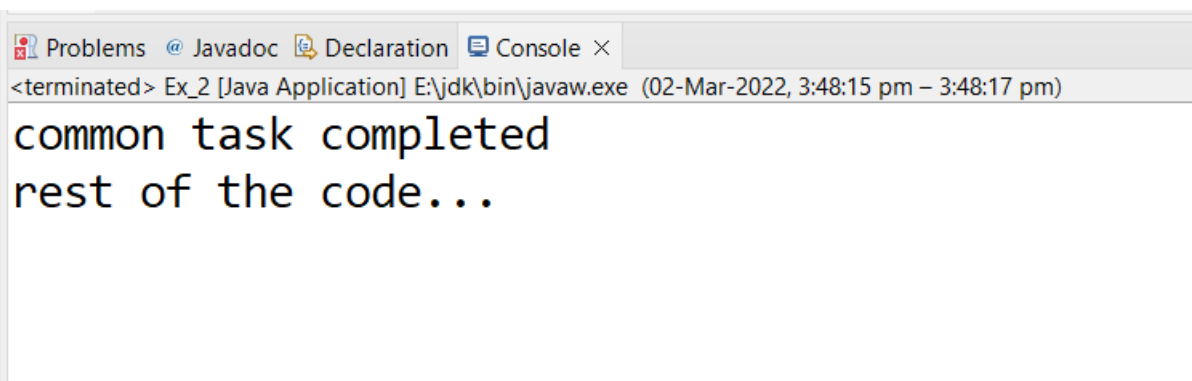
When any try block does not have a catch block for a particular exception, then the catch block of the outer (parent) try block are checked for that exception, and if it matches, the catch block of outer try block is executed.

```

public class Ex_2 {
    public static void main(String[] args) {
        try{
            try {
                int a;
                a=30/0;
            }

```

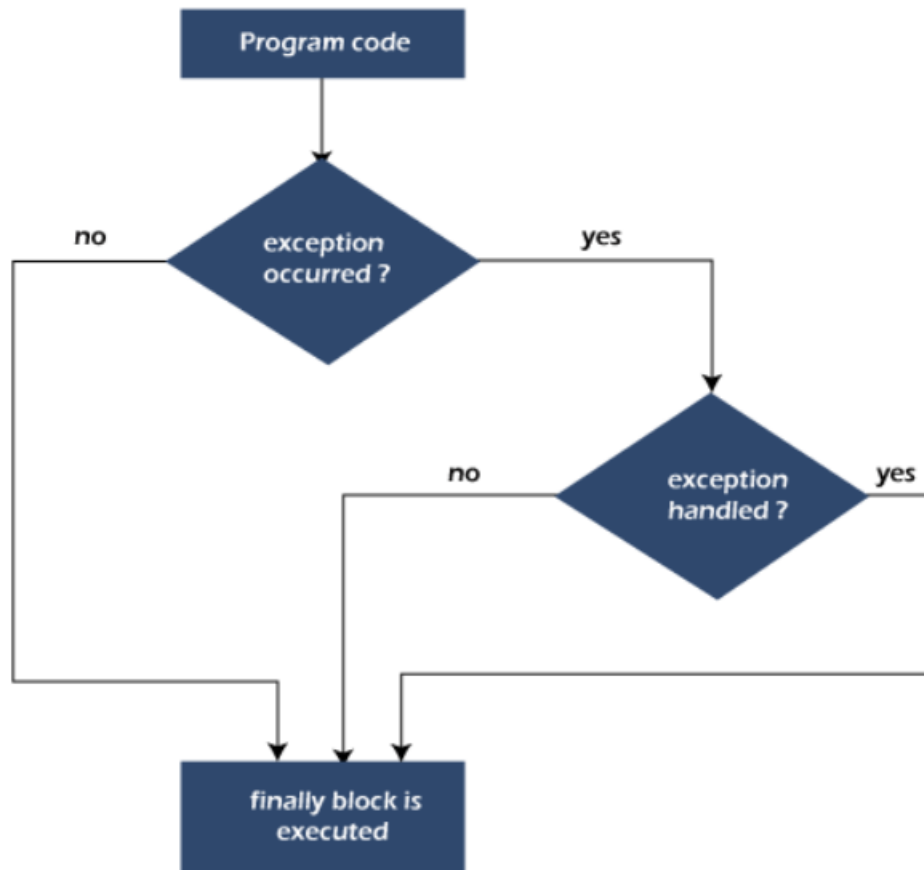
```
    }  
  
    catch(ArrayIndexOutOfBoundsException  
e){System.out.println("task1 is completed");}  
    }  
    catch(Exception  
e){System.out.println("common task  
completed");}  
    System.out.println("rest of the  
code...");  
    }  
}
```



The screenshot shows an IDE window with a tab labeled 'Console'. The console output displays the results of a Java application execution. The first line is '<terminated> Ex_2 [Java Application] E:\jdk\bin\javaw.exe (02-Mar-2022, 3:48:15 pm - 3:48:17 pm)'. Below this, the application's output is shown: 'common task completed' followed by 'rest of the code...' on the next line.

```
<terminated> Ex_2 [Java Application] E:\jdk\bin\javaw.exe (02-Mar-2022, 3:48:15 pm - 3:48:17 pm)  
common task completed  
rest of the code...
```

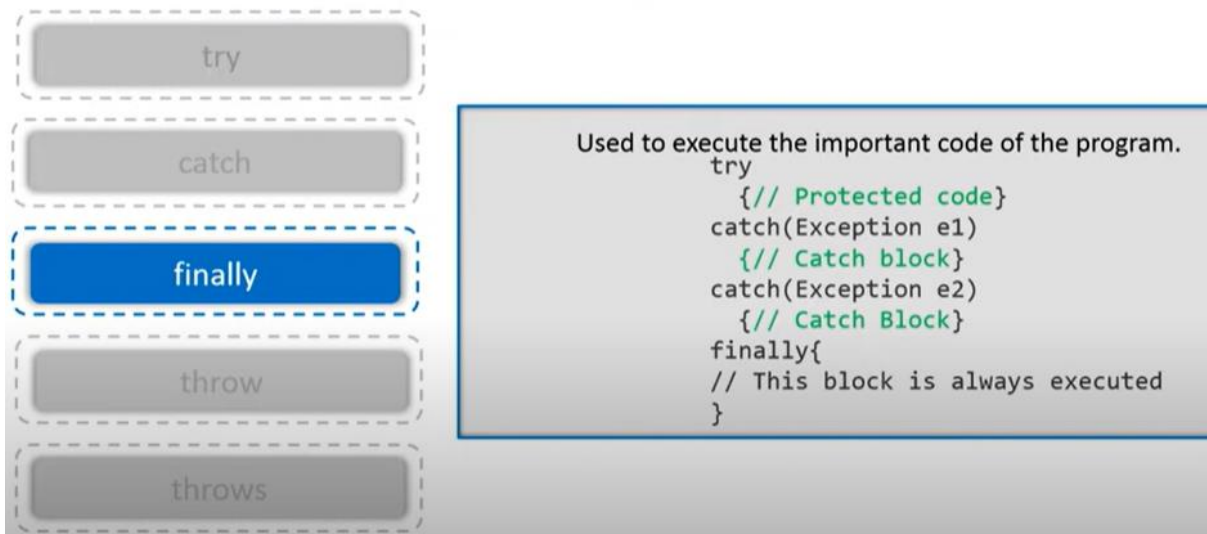

Java finally block



Why use Java finally block?

- finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
- The important statements to be printed can be placed in the finally block.

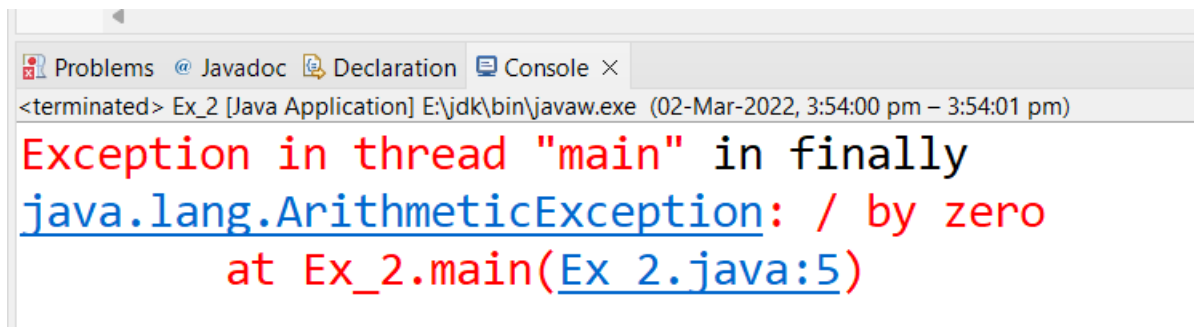
Exception Handling Methods



Rule: For each try block there can be zero or more catch blocks, but only one finally block.

When an exception occurs but is not handled by the catch block

```
public class Ex_2 {
    public static void main(String[] args) {
        try{
            int a;
            a=30/0;}
        finally {
            System.out.println("in finally");
        }
    }
}
```



```
Problems @ Javadoc Declaration Console ×
<terminated> Ex_2 [Java Application] E:\jdk\bin\javaw.exe (02-Mar-2022, 3:54:00 pm – 3:54:01 pm)
Exception in thread "main" in finally
java.lang.ArithmeticException: / by zero
    at Ex_2.main(Ex_2.java:5)
```

the finally block is executed after the try block and then the program terminates abnormally.

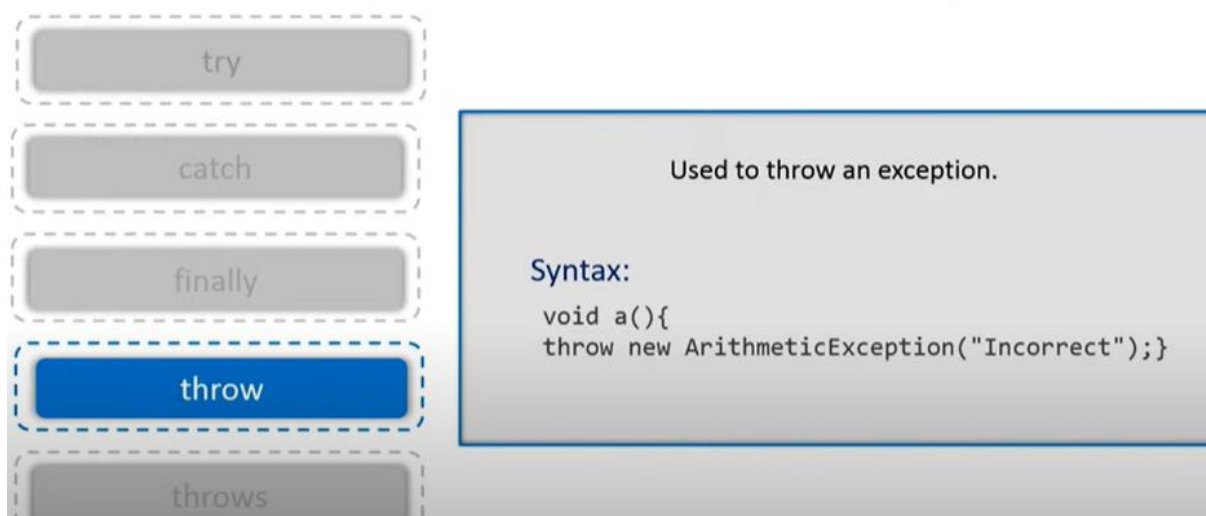
Java throw keyword

The Java throw keyword is used to throw an exception explicitly.

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception.

Exception Handling Methods



try

catch

finally

throw

throws

Used to throw an exception.

Syntax:

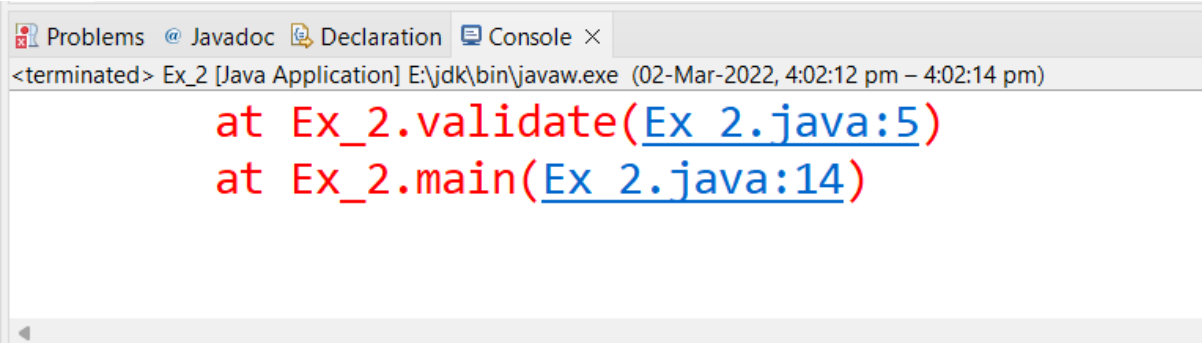
```
void a(){
    throw new ArithmeticException("Incorrect");}
```

```
public class Ex_2 {
```

```

        public static void validate(int
age) {
            if(age<18) {
                //throw Arithmetic
exception if not eligible to vote
                throw new
ArithmeticException("Person is not eligible
to vote");
            }
            else {
                System.out.println("Person
is eligible to vote!!");
            }
        }
        //main method
        public static void main(String
args[]){
            //calling the function
            validate(13);
            System.out.println("rest of
the code...");
        }
    }

```



```

Problems  Javadoc  Declaration  Console ×
<terminated> Ex_2 [Java Application] E:\jdk\bin\javaw.exe (02-Mar-2022, 4:02:12 pm – 4:02:14 pm)
    at Ex_2.validate(Ex 2.java:5)
    at Ex_2.main(Ex 2.java:14)

```

Java throws keyword

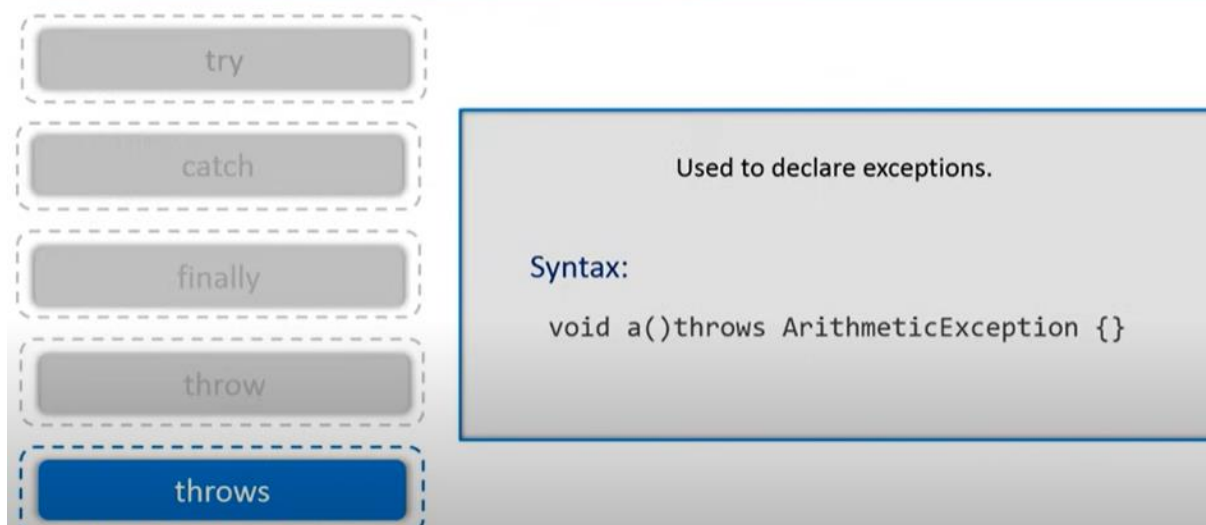
The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

Which exception should be declared?

Ans: Checked exception only, because:

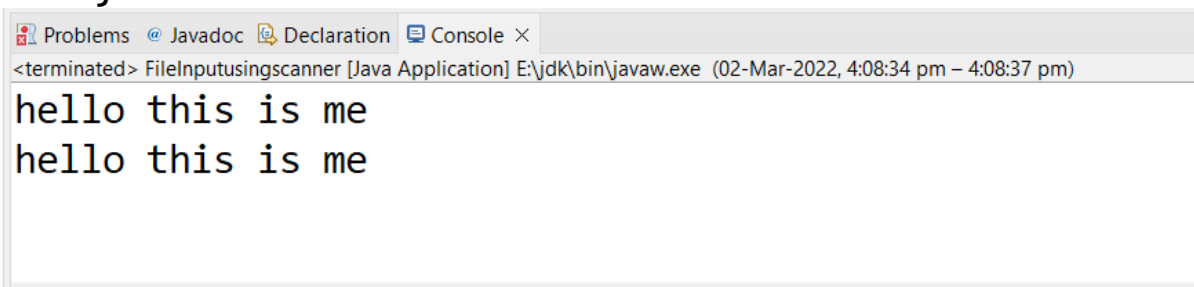
- **unchecked exception:** under our control so we can correct our code.
- **error:** beyond our control. For example, we are unable to do anything if there occurs `VirtualMachineError` or `StackOverflowError`.

Exception Handling Methods



- In case we declare the exception, if exception does not occur, the code will be executed fine.
- **import** `java.io.FileInputStream;`
- **import** `java.io.FileNotFoundException;`
- **import** `java.io.IOException;`
- **import** `java.util.Scanner;`
-

- `public class` FileInputusingscanner {
-
- `public static void` main(String[] args)
- `throws` IOException {
- `// TODO` Auto-generated method stub
- FileInputStream `f` = `new`
- FileInputStream("new.txt");
- Scanner `s` = `new` Scanner(`f`);
- `while`(`s.hasNext()`) {
-
- System.`out`.println(`s.nextLine()`);
- }
-
- }
-
- }



The screenshot shows an IDE window with a tab labeled 'Console'. The console output displays two lines of text: 'hello this is me' followed by a new line and 'hello this is me' again. The window title bar indicates the application is 'FileInputusingscanner [Java Application]' running at 'E:\jdk\bin\javaw.exe' on '02-Mar-2022, 4:08:34 pm - 4:08:37 pm'.

- **In case we declare the exception and the exception occurs, it will be thrown at runtime because throws does not handle the exception.**
- **When there is no such file named new.txt**

```
Problems Javadoc Declaration Console x
<terminated> FileInputusingscanner [Java Application] E:\jdk\bin\javaw.exe (02-Mar-2022, 4:09:32 pm - 4:09:34 pm)
Exception in thread "main" java.io.FileNotFoundException: new.txt (The system cannot find the file specified)
at java.base/java.io.FileInputStream.open0(Native Method)
at java.base/java.io.FileInputStream.open(FileInputStream.java:216)
at java.base/java.io.FileInputStream.<init>(FileInputStream.java:157)
```

Throw vs Throws

1. Used to explicitly throw an Exception.
2. Checked Exceptions cannot be propagated using throw only.
3. Followed by an instance.
4. Used within a method.
5. Cannot throw multiple exceptions.



1. Used to declare an Exception.
2. Checked Exceptions can be propagated.
3. Followed by a class.
4. Used with a method Signature.
5. Can declare Multiple Exceptions.

Final vs Finally vs Finalize

1. Keyword
2. Applies restrictions on class, method and variable.
3. final class cant be inherited, method cant be overridden & the variable value cant be changed

1. Block
2. Used to place an important code
3. It will be executed whether the exception is handled or not.

1. Method
2. Used to perform clean-up processing just before the object is garbage collected

Java Custom Exception

In Java, we can create our own exceptions that are derived classes of the **Exception** class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

1st step your user define exception must extends exception class

//2nd step exception is thrown using throw keyword

```
class MyException extends Exception{}
```

//1st step your user define exception must extends exception class

```
public class customexception {
```

```
    public static void main(String[] args) {  
        try {  
            throw new MyException();
```

//2nd step exception is thrown using throw keyword

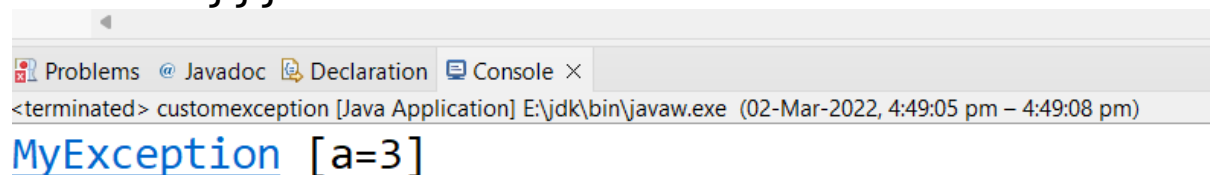
```
        } catch (MyException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }
```

```
    }
```


Another example

```
class MyException extends Exception{
    public int a;
    public MyException(int a) {
        this.a = a;
    }
    @Override
    public String toString() {
        return "MyException [a=" + a + "]";
    }
}

public class customexception {
    public static void main(String[] args) {
        try {
            throw new MyException(3);
        } catch (MyException e) {
            System.out.println(e);
        }
    }
}
```



The screenshot shows an IDE window with tabs for Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the output of the program: `<terminated> customexception [Java Application] E:\jdk\bin\javaw.exe (02-Mar-2022, 4:49:05 pm - 4:49:08 pm)` followed by `MyException [a=3]`.

Let's see a simple example of Java custom exception. In the following code, constructor of `InvalidAgeException` takes a string as an argument. This string is passed to constructor of parent class `Exception` using the `super()` method. Also the constructor of `Exception` class can be called without using a parameter and calling `super()` method is not mandatory.

```
// class representing custom exception
class InvalidAgeException extends Exception
{
    public InvalidAgeException (String str)
    { // calling the constructor of parent
      Exception
        super(str);
    }
}
```

```
}  
    // class that uses custom exception  
InvalidAgeException  
class TestCustomException1  
{  
    // method to check the age  
    static void validate (int age) throws  
InvalidAgeException{  
        if(age < 18){  
            // throw an object of user defined  
exception  
                throw new InvalidAgeException("age  
is not valid to vote");  
        }  
        else {  
            System.out.println("welcome to  
vote");  
        }  
    }  
  
    // main method  
    public static void main(String args[])  
    {  
        try  
        {  
            // calling the method  
            validate(13);  
        }  
        catch (InvalidAgeException ex)  
        {  
            System.out.println("Caught the  
exception");  
        }  
    }  
}
```

```
        // printing the message from
InvalidAgeException object
        System.out.println("Exception
occured: " + ex);
    }

    System.out.println("rest of the
code...");
}
}

}
```