Object - Oriented Programming 2

Week 1: Course Overview, Introduction to Exceptions, try and catch block, multiple catch and nested try blocks

By Elubu Joseph - MSc.IS

Lecturer

Department of Information Technology

Kumi University

Email: josebulinda@gmail.com

jose@kumiuniversity.ac.ug

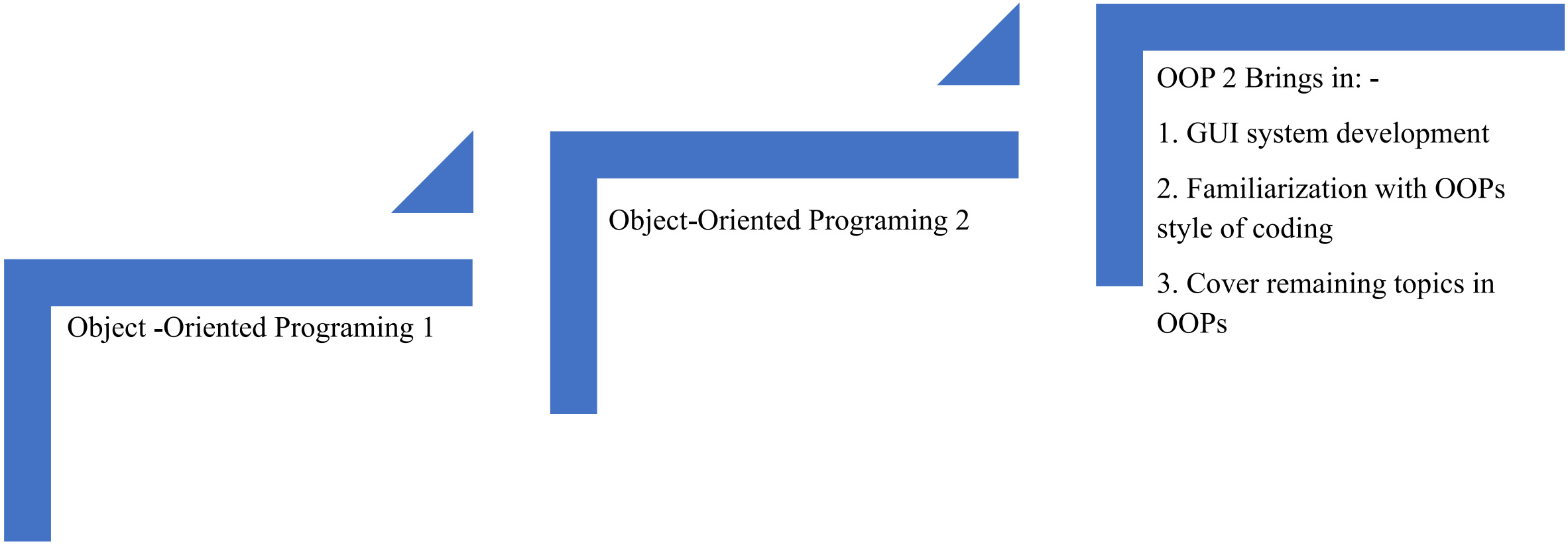
Agenda

1. Course Overview
2. Introduction to Exceptions; Difinitions, Classification of Exceptions, Exception handling etc.
3. try and catch block; Multiple catch blocks,nested try blocks

Course Overview

1. Course Description
2. Course Objectives
3. Course requirements
4. Course length
5. Text Books and References

Course Overview-Course Description



Objected-Oriented Programming 2 builds on Object-Oriented Programming I and it utilizes coding skills acquired in Objected-Oriented Programming 1 course to help the learner build advance software systems, using techniques such as use of graphical user interface tools provided by various Integrated Development Environment - IDE systems.

Course Overview-Course Objectives

The objective of this course is:-

- To provide a familiarity with the syntax, class hierarchy acquired from Object Oriented Programming 1 through the provision of more practical time.
- To develop an understanding of the principles of the object-oriented paradigm based on graphical User Interface software development and use of databases in software development
- To dig deeper into the remaining approaches to or topics object-oriented style of system development

Course Overview-Requirements

For the learner to flow well in this course, he/she should have studied Object-Oriented Programming 1 course on which concepts learnt, this course builds on.

Course Overview - Course Length

As stipulated in the syllabus, this course is designed to run for 13 weeks. For details about the break down of the content, kindly view the syllabus.

Text Books

You can use the following books,

1. Beginning Java Programming, Deepak,V., et al (2015),: The Object-Oriented Approach, John Wiley & Sons
2. Java Programming 8th Edition, Cengage Learning, Joyce ,F.l (2015)
3. Beginning Programming with Java for Dummies3th Edition, Burd,B. (2012); John Wiley & Sons 128
4. Object Oriented Computer Systems Engineering, Derrick, M., David, E., Peter, G., Colin T. (2012),, Springer Science & Business Media, ISBN
5. Introduction to Object-oriented Programming with JAVA4th Edition,Tata (2011). McGraw Hill Education

Introduction to Exceptions

Gen. Definition 1 : An exception is a person or thing that is excluded from a general statement or does not follow a rule.

Definition 2: An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.(studytonight.com, n.d)

Examples of Exceptional Situations

During the execution of a program, it may experience abnormal or exceptional conditions. As a result of this, the system may crash. An exception may occur due to a number of reasons. Some of these include: -

1. A file that needs to be opened can't be found.
2. A client has entered invalid information.
3. A system association has been lost amidst correspondences or the JVM has used up all the available memory.

Some of these special cases are created by **client mistake**, others by **developer blunder**, and others **by physical assets** that have fizzled into your code in some way. (Sanderson, n.d.)

Classifications of Exceptions

To see how exception handling works in Java, we have to comprehend the three classifications of exceptions:

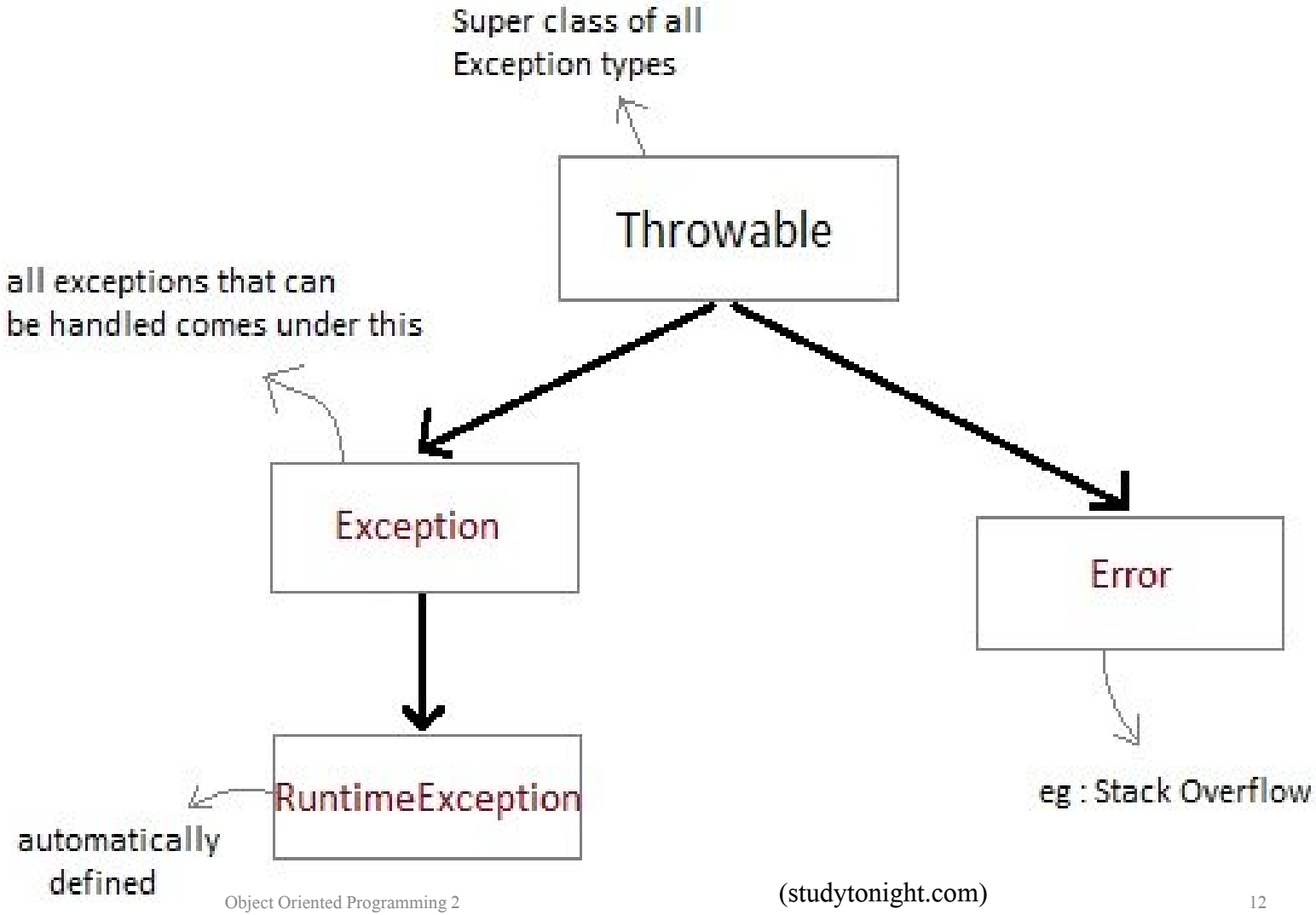
1. Errors: are typically ignored in code because one can rarely do anything about an error. For **example**, if stack overflow occurs, an error will arise. This type of error cannot be handled in the code.

2. **Unchecked exceptions** are the class that extends RuntimeException class. Unchecked exception are ignored at compile time and checked at runtime. For **example** : ArithmeticException, NullPointerException, ArrayIndexOutOfBoundException. Unchecked exceptions are checked at runtime.

3. **Checked Exceptions:** It is a special case that is regularly a **client mistake** or an issue that can be predicted by the developer. Case in point, if a file is to be opened, yet the file can't be found, an exception of this type happens. These special cases can't just be disregarded at the time of compilation and dry runs (Sanderson, n.d.)

Java Exception class Hierarchy

All exception types are subclasses of class **Throwable**, which is at the top of exception class hierarchy.



(studytonight.com)

Java Exception class Hierarchy+

Exception class is for exceptional conditions that the program should catch. This class is extended to create user specific exception classes.

RuntimeException is a subclass of Exception. Exceptions under this class are automatically defined for programs.

Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

Danger of exceptions

The problem with the exception is that, it terminates the program and skip the rest of the execution. This means, if a program has 110 lines of code and at line 10 an exception occur then program will terminate immediately by skipping execution of 100 lines of code.

To handle this problem, we use exception handling techniques to avoid program termination and continue the execution by skipping exception code.

Java exception handling provides a meaningful message to the user about the issue rather than a system generated message, *which* ***may not be understandable to a user****.*

Example -Uncaught Exception

As you may have known, division by zero is not allowed. However there are scenarios where users end up dividing values by zero, which is an exception hence termination of the program if not caught.

E.g. 200/0 =?

Will throw ArithmeticException

```
class UException{
public static void main(String args[])
{
int k = 0; int l = 200;
// Divide by zero, will lead to
exception
int b = l/k;
System.out.println(b);
}
}
```

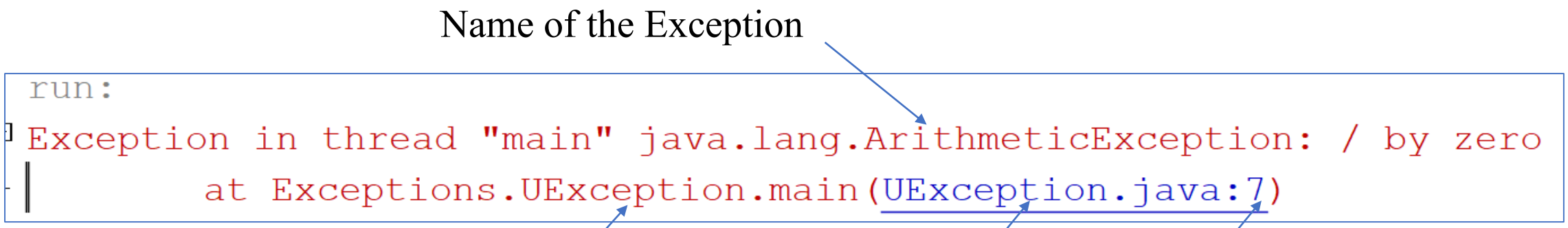
ArithmeticException Generated

```
2   package Exceptions;
3   public class UException {
4       public static void main(String[] args) {
5           int k = 0; int l = 200;
6       // Divide by zero, will lead to exception
7       int b = l/k;
8       System.out.println(b);
9       }
10  }
```

```
run:
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Exceptions.UException.main(UException.java:9)
C:\Users\joseb\AppData\Local\NetBeans\Cache\13\executor-snippets\run.xml:111: The following error occurred while executing this line:
C:\Users\joseb\AppData\Local\NetBeans\Cache\13\executor-snippets\run.xml:68: Java returned: 1
BUILD FAILED (total time: 1 second)
```

Explanation

Uncaught exception by the programmer above led to program termination at runtime, hence the Java run-time system auto constructed an exception and then threw it. hence the default handler (JVM) handled the exception and printed the details of the exception on the terminal.



Exception Handling

A Java Exception is an object that describes the exception that occurs in a program.

When an exceptional events occurs in java, an exception is said to been thrown. The code that's responsible for doing something about the exception is called an **exception handler,** which then deals with uncaught exceptions by the programmer**.**

Exception Handling+

Java provides controls to handle exception in the program. These controls are:.

1. try : It is used to enclose the suspected code.
2. catch: It acts as exception handler.
3. finally: It is used to execute necessary code.
4. throw: It throws the exception explicitly.
5. throws: It informs for the possible exception.

try and catch in Java

try and **catch** are Java **keywords** and used for **exception handling**.

The **try** block is used to enclose the **suspected** code. Suspected code is a code that may raise an exception during program execution.

For example, if a code raise ArithmeticException due to divide by zero then we can wrap that code into the try block.

try block example

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

    int k = 0;
    int l = 200;
    int b = l/k;//Exception here
    System.out.println(b);

    }
}
}
```

The catch block

The **catch** block also known as **handler** is used to handle the exception. It handles the exception thrown by the code enclosed inside the try block. Try block must provide a catch handler or a finally block. We will discuss about finally block in our next tutorials.

The catch block must be used after the try block only. We can also use multiple catch block with a single try block. (studytonight.com)

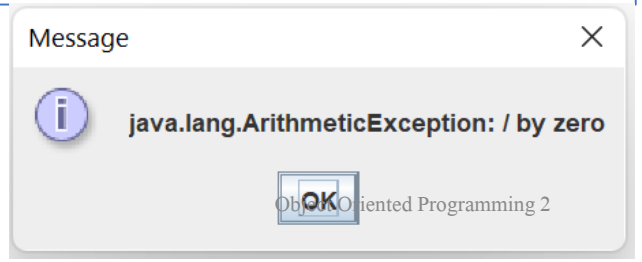
try and catch block example

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

      int k = 0;
      int l = 200;
      int b = l/k;//Exception here
      System.out.println(b);
      }
      catch(ArithmeticException e){
            JOptionPane.showMessageDialogBox(null, e);
      }
}
}
```

```
2   package Exceptions;
3   import javax.swing.JOptionPane;
4   public class tryBlock{
5       public static void main(String[] args) {
6           // puting suspect code inside try block
7           try{
8           int k = 0; int l = 200;
9
10          int b = l/k;
11          System.out.println(b);
12          }
13          catch(ArithmeticException e){
14              JOptionPane.showMessageDialog(null, e);
15          }
16      }
17  }
```

OUTPUT



The code above will still generate and ArethmeticException because of divide by zero code inside the try block. Therefore there will be NO output at line 11, But the difference here is that the exception will be thrown from the try block then caught by the ctach block thus line 14 will be executed instead so the program will not terminate before completion.

try block without error in the code

Without error in the **try** block, the **catch block** will not handle any Exceptions.

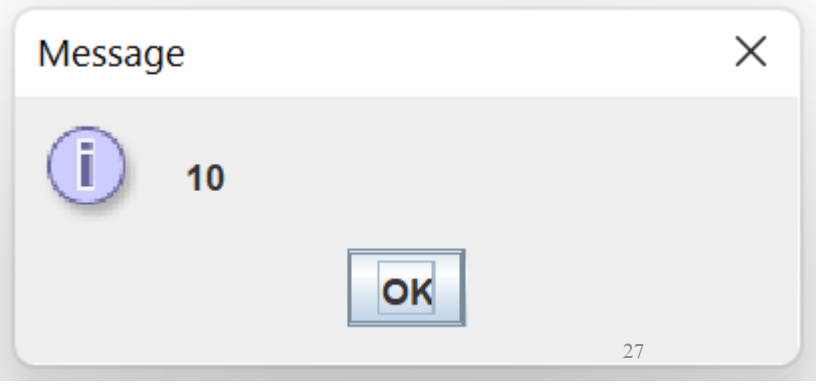
```
class tryAndCatchBlock{
public static void main(String args[]) {
      try{

      int k = 20;
      int l = 200;
      int b = l/k;//Exception here
      JOptionPane.showMessageDialog(null, b);
      }
      catch(ArithmeticException e){
            JOptionPane.showMessageDialogBox(null, e);
      }
}
}
```

try block without error in the code

```
1   package Exceptions;
2   import javax.swing.JOptionPane;
3   public class tryBlock{
4       public static void main(String[] args) {
5           // puting suspect code inside try block
6           try{
7           int k = 20; int l = 200;
8
9           int b = l/k;
10          JOptionPane.showMessageDialog(null, b);
11          }
12          catch(ArithmeticException e){
13              JOptionPane.showMessageDialog(null, e);
14          }
15      }
16  }
```

OUTPUT



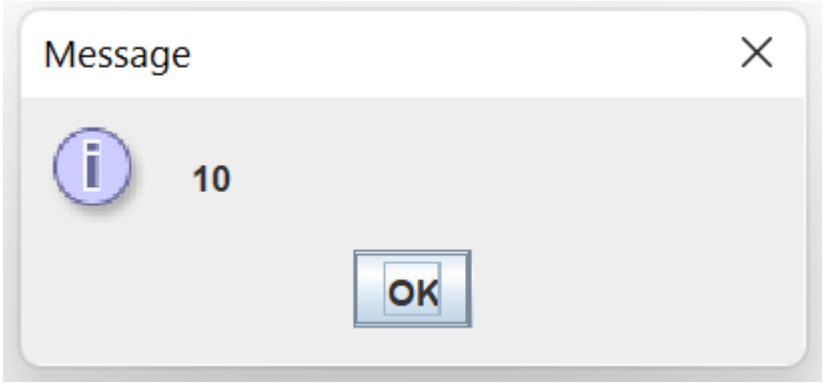
Explanation

Note that NO exception was thrown by the above program as we were dividing 200/20 inside **try** block.

Thus Line 10 executed and the output was seen as the program ran normally.

```
9            int b = 1/k;
10           JOptionPane.showMessageDialog(null, b);
11           }
```

OUTPUT: 10



Working with Multiple catch blocks

A try block can be followed by multiple catch blocks.

It means we can have any number of catch blocks after a single try block. If an exception occurs in the guarded code (try block) the exception is passed to the first catch block in the list.

If the exception type matches with the first catch block it gets caught, if not the exception is passed down to the next catch block. This continue until the exception is caught or falls through all catches.

Multiple Catch Syntax

To declare multiple catch handler, we can use the following syntax.

```
try
   {
    // suspected code
   }
   catch(Exception1 e){
     // handler code
   }
   catch(Exception2 e){
   // handler code
   }
```

Note. The code above has two **catch** blocks

Example 1 Multiple Catch blocks

Now lets see an example to implement the multiple **catch** block that are used to **catch** possible exception.

The multiple **catch** blocks are useful *when we are not sure about the type of exception* during program execution.

In this example, we are trying to fetch integer value of an Integer object. But due to bad input, it throws NumberFormatException.

Examples for Multiple Catch blocks+

```
class multipleCatchBlock{

  public static void main(String[] args) {
    try
    {
      Integer in = new Integer("abc");
      in.intValue();
    }
    catch (ArithmeticException e)
    {
      System.out.println("Arithmetic " + e);
    }
    catch (NumberFormatException e)
    {
      System.out.println("Number Format Exception " + e);
    }
  }
}
```

In the above example, we used multiple catch blocks and based on the type of exception, second catch block is executed.

Studytonight.com

```java
package Exceptions;
public class multipleCatchBlock {
    public static void main(String[] args) {
        // Working with multiple ctach block
        try
    {
      Integer in = new Integer("abc");
      in.intValue();
    }
    catch (ArithmeticException e)
    {
      System.out.println("Arithmetic " + e);
    }
    catch (NumberFormatException e)
    {
      System.out.println("Number Format Exception " + e);
    }
    }
```

```
run:
Number Format Exception java.lang.NumberFormatException: For input string: "abc"
BUILD SUCCESSFUL (total time: 0 seconds)
```

Example 2: Multiple Exception+

Lets understand the use of multiple catch handler by **one more example,** here we are using three catch handlers in catch the exception.

```
public class multicatchBlock2{
  public static void main(String[] args){
    try{
      int a[]=new int[10];
      System.out.println(a[20]);
    }
    catch(ArithmeticException e){
      System.out.println("Arithmetic Exception --> "+e);
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("ArrayIndexOutOfBounds Exception --> "+e);
    }
    catch(Exception e)  {
      System.out.println(e);
    }
}
}
```

Note that we would like to print out the element at index 20 of array a[] and yet the said array has only 11 indicies. This throws ArrayIndexOutOfBoundsException.

```java
package Exceptions;
public class multicatchBlock2 {
    public static void main(String[] args) {
        try{
      int a[]=new int[10];
      System.out.println(a[20]);
    }
    catch(ArithmeticException e){
      System.out.println("Arithmetic Exception --> "+e);
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("ArrayIndexOutOfBounds Exception --> "+
              e);
    }
    catch(Exception e)  {
      System.out.println(e);
    }
    }
}
```

Note that index 20 is out of bounds for array a[].

OUTPUT

```
run:
ArrayIndexOutOfBounds Exception --> java.lang.ArrayIndexOutOfBoundsException: Index 20 out of bounds for length 10
BUILD SUCCESSFUL (total time: 0 seconds)
```

Example of Unreachable Catch Block

While using multiple **catch** statements, it is important to remember that sub classes of class **Exception** inside **catch** block must come before any of their super classes otherwise it will lead to compile time error.

This is because in Java, if any code is unreachable, then it gives compile time error.

Unreachable Exception

```
class unReachableExcep{
  public static void main(String[] args){
    try{
      int Kumu[]={1,2};
      Kumu[2]=3/0;
    }
    catch(Exception e)     /*/This block handles all Exception*
    {
      System.out.println("Generic exception");
    }
    catch(ArrayIndexOutOfBoundsException e) /*/This block is unreachable*/
    {
      System.out.println("array index out of bound exception");
    }
  }
}
```

Exception e in the first **catch** block must be the last in the hierarchy because it is a super class to the rest. Meanwhile **ArrayIndexOutOfBoundsException** was suppose to be caught, it is unreachable since it is located in the second catch block after Exception e catch block.

```
1   package Exceptions;
2   public class unReachableExcep {
3       public static void main(String[] args) {
4           try{
5         int Kumu[]={1,2};
6         Kumu[2]=3/0;
7       }
8       catch(Exception e)     /*This block handles all Exception*/
9       {
10                                          neric exception");
11
12      catch(ArrayIndexOutOfBoundsException e){
13        System.out.println("array index out of bound exception");
14      }
15
16      }
17
18  }
```

exception ArrayIndexOutOfBoundsException has already been caught
---
(Alt-Enter shows hints)

Note the compile time error:

Exception ArrayIndexOutOfBoundsException has already been caught.

Right Hierarchy of Exceptions

```
1   package Exceptions;
2   public class unReachableExcep2 {
3       public static void main(String[] args) {
4           try{
5           int Kumu[]={1,2};
6           Kumu[2]=3/0;
7       }
8       catch(ArrayIndexOutOfBoundsException e)
9       {
10          System.out.println("Generic exception");
11      }
12      catch(Exception e){
13          System.out.println("array index out of bound exception");
14      }
15      }
16  }
```

```
run:
array index out of bound exception
BUILD SUCCESSFUL (total time: 1 second)
```

Nested try Block

try block can be **nested** inside another block of **try**. Nested **try** block is used when a part of a block may cause one error while entire block may cause another error.

In case if inner **try** block does not have a **catch** handler for a particular exception then the outer **try catch block** is checked for match.

Example of Nested try statement

```
class nestedTryBlock{
  public static void main(String[] args){
    try{
      int ku[]={5,0,1,60};
      try{
        int x = ku[3]/ku[1];
      }
      catch(ArithmeticException ae){
        System.out.println("divide by zero");
      }
      ku[4]=3;
    }
    catch(ArrayIndexOutOfBoundsException e){
      System.out.println("array index out of bound exception");
    }
  }
}
```

Note that inner try block thows ArithmeticException while outer try block thows ArrayIndexOutOfBoundsException due to an atempt to divide 60 by 0 and Changing the value of index 4 that does not exist in array ku[].

Example of Nested try statement

```
1   package Exceptions;
2   public class nestedTryBlock {
3       public static void main(String[] args) {
4         try{
5         int ku[]={5,0,1,60};
6         try{
7           int x = ku[3]/ku[1];
8         }
9         catch(ArithmeticException ae){
10          System.out.println("divide by zero");
11        }
12        ku[4]=3;
13      }
14      catch(ArrayIndexOutOfBoundsException e){
15        System.out.println("array index out of bound exception");
16      }
17      }
18  }
```

OUTPUT

```
run:
divide by zero
array index out of bound exception
BUILD SUCCESSFUL (total time: 1 second)
```

Summary

Summary

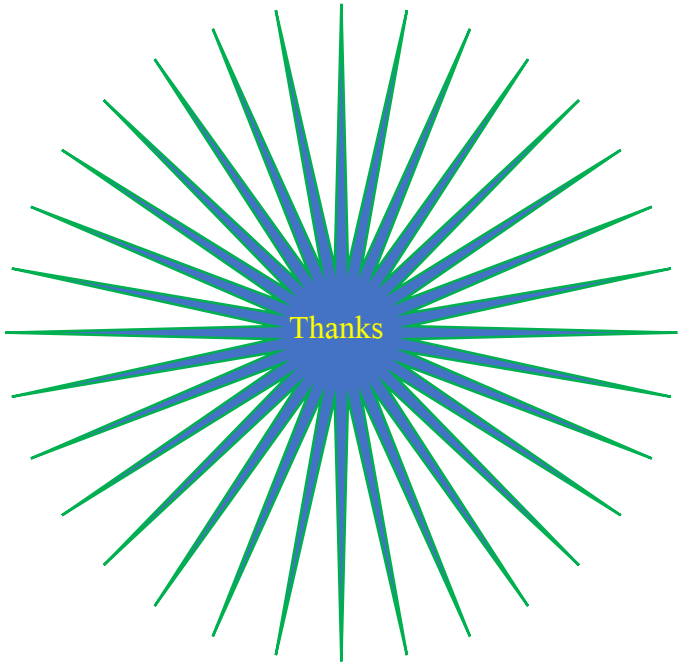
1. Course Overview-Course Descrioption, Objectives, Requirements, Text Books and refrerences
2. Introduction to Exceptions; Definitions, Classification of Exceptions, Exception handling etc.
3. try and catch block; Multiple catch blocks,nested try blocks

Important points to Remember

Important points to Remember

1. If you do not explicitly use the try catch blocks in your program, java will provide a default exception handler, which will print the exception details on the terminal, whenever exception occurs.
2. Super class **Throwable** overrides **toString()** function, to display error message in form of string.
3. While using multiple **catch** block, always make sure that sub-classes of **Exception** class comes before any of their super classes. Else you will get compile time error.
4. In **nested try block**, the inner try block uses its own catch block as well as catch block of the outer try, if required.
5. Only the object of **Throwable** class or its subclasses can be thrown.

Thank you for Listening



References

Java For Beginners Sanderson, S. (n.d.). ***A Simple Start To Java Programming***

Java exception handling. Studytonight.com. (n.d.). Retrieved September 8, 2022, from https://www.studytonight.com/java/exception-handling.php