Object - Oriented Programming 2

Week 2. Java try with Resource Statement, Java throw, throws and finally Keyword

By Elubu Joseph - MSc.IS

Lecturer

Department of Information Technology

Kumi University

Email: josebulinda@gmail.com

jose@kumiuniversity.ac.ug

Agenda

1. Java **try** with Resource Statement,
2. Java **throw**, **throws** and **finally** Keyword
3. Dealing with user defined Exception Classes
4. Method Overriding with Exception Handling,
5. Chained Exceptions

Java **try** with Resource Statement,

try with resource add another way to exception handling with resources management. This is a new feature of Java that was introduced in Java 7 and further improved in Java 9. It is also referred as **automatic resource management**. It close resources automatically by using AutoCloseable interface.

So, what is a resource in this case? A resource is an object that is used in program and must be closed after the program is finished

A Resource is anything like: file, connection etc and we don't need to explicitly close these, JVM will do this automatically.

For example, suppose we run a JDBC program to connect to the database, then we have to create a connection and close it at the end of task as well. But while using **try-with-resource feature, we don't need to close the connection**, JVM will do this automatically by using AutoCloseable interface. (studytonight.com,n.d)

Try with Resource Syntax

Note! The use of parenthesis introduced for the try statement at this point. Where more than one resource may be passed

```
try(resource-specification(more than one
resource may be passed here)){
    //use the resource
}
catch()
{
    // handler code
}
```

Try with Resource Syntax+

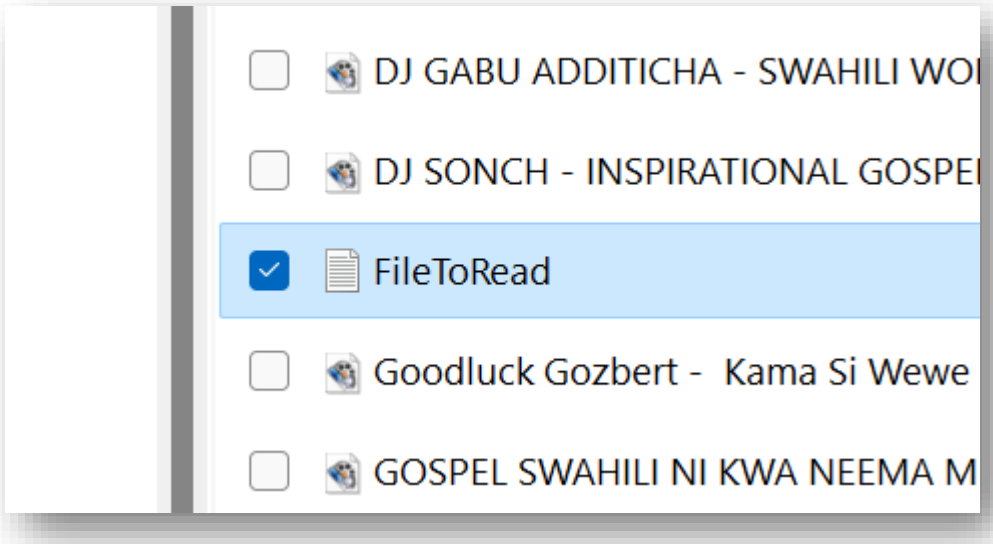
This **try statement** contains a **parenthesis** in which one or more resources is declared. Any object that implements java.lang.AutoCloseable or java.io.Closeable, can be passed as a parameter to **try statement**.

A resource is an object that is used in program and must be closed after the program is finished. The **try-with-resources statement** ensures that each resource is closed at the end of the statement of the try block. We do not have to explicitly close the resources.

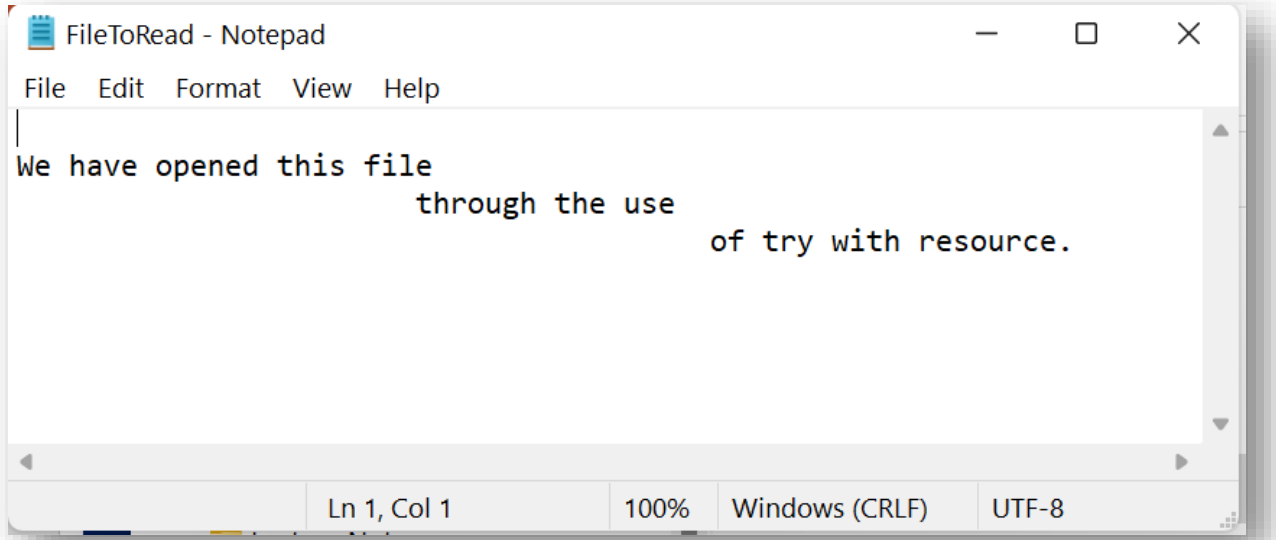
Example try with Resource Statement

Here, we are using **try-with-resource** to open and close a file named **FileToRead** found in our local disk (D:) without using the **close() method** to close the file connection.

File to be opened inside the Disk (D:)



The File opened



Example try with Resource Statement+

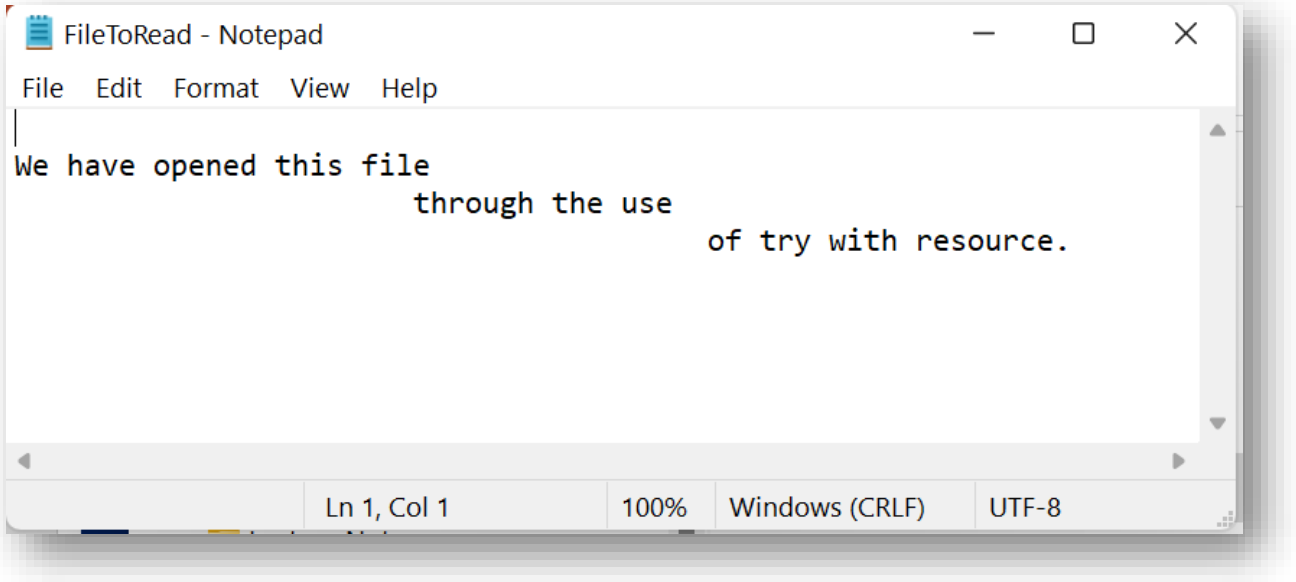
```
import java.io.*;
class tryWithResource{
public static void main(String[] args){
      try(BufferedReader br = new BufferedReader(new
      FileReader("D:\\FileToRead.txt"))){
      String read;
      while((read = br.readLine())!= null){
            System.out.println(read);
      }
}
catch(IOException ie){
System.out.println("I/O Exception "+ie);
}
} }
```

Try with Resource Syntax+++ Code

```
3  import java.io.*;
4  import java.io.IOException;
5  public class tryWithResource {
6      public static void main(String[] args) {
7          try(BufferedReader br = new BufferedReader(new
8          FileReader("D:\\FileToRead.txt"))){
9          String read; while((read = br.readLine()) != null){
0          System.out.println(read);
1          }
2      }
3      catch(IOException ie) {
4      System.out.println("I/O Exception "+ie);
5      }
6      }
7  }
```

The file opened by Notepad and by the code.

The File opened in Notepad



The File opened by the Code

```
run:

We have opened this file
                        through the use
                                        of try with resource.
```

Java throw, throws and finally Keyword

Throw, throws and finally

are the keywords in Java that are used in exception handling.

So what is the difference between throw and throws?

The **throw** keyword is used to throw an exception while **throws** is used to declare the list of possible exceptions with the method signature.

Whereas **finally** block is used to execute essential code, **specially to release the occupied resources.**

Lets discuss each in details with the examples.

Difference between throw and throws

throw keyword	**throws keyword**
throw is used to throw an exception explicitly.	throws is used to declare an exception possible during its execution.
throw is followed by an instance of Throwable class or one of its sub-classes.	throws is followed by one or more Exception class names separated by commas.
throw is declared inside a method body.	throws is used with method signature (method declaration).
We cannot throw multiple exceptions using throw keyword.	We can declare multiple exceptions (separated by commas) using throws keyword.

Java Throw

The throw keyword is used to throw an exception explicitly. Only object of Throwable class or its sub classes can be thrown. Program execution stops on encountering **throw** statement, and the closest **catch statement** is checked for matching type of exception.

Syntax:

```
throw ThrowableInstance
```

Creating Instance of Throwable class

We use the **new** operator to create an instance of class Throwable,

```
new NullPointerException("Tester");
```

Note the **new** operator is used. This constructs an instance of NullPointerException with name **Tester**.

Example throw Exception program

In this example, we are **throwing ArithmeticException** explicitly by using the throw keyword that will be handled by the catch block.

```
class throwExcep{
	public static void main(String args[]){
	avg();
	}
	static void avg(){
	try {
		throw new ArithmeticException("String not
allowed");
	}
	catch(ArithmeticException e){
		System.out.println("Exception caught");
	}
	}
}
```

The code

```
2   package Exceptions;
3   public class throwExcep {
4       public static void main(String[] args) {
5           avg();
6       }
7       static void avg(){
8           try {
9           throw new ArithmeticException("String not allowed");
10          }
11          catch(ArithmeticException e){
12                  System.out.println("Exception caught");
13          }
14          }
15  }
```

OUTPUT

```
run:
Exception caught
```

Note an error here

In the above example the avg() method throw an instance of ArithmeticException, which is successfully handled using the catch statement and thus, the program prints the output "Exception caught".

Java throws Keyword

The throws keyword is used to **declare the list of exception** that a method may throw during execution of a program.

Any method that is capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions are to be handled.

A method can do so by using the **throws** keyword.

Syntax:

Java throws Keyword

Syntax:

```
type method_name(List of Parameter) throws exception_list{

// definition of method

}
```

Example throws Keyword

Here, we have a method that can throw **Arithmetic Exception**, so we mentioned that with the method declaration and catch that using the catch handler in the main method.

```
class throwsExcep{
static void check() throws ArithmeticException{
      System.out.println("Inside check function");
      throw new ArithmeticException("String");
}

public static void main(String args[]){
      try {
            check();
      }
      catch(ArithmeticException e){
      System.out.println("caught" + e);
      }
} }
```

The Code

```
2   package Exceptions;
3   public class throwsExcep {
4       static void check() throws ArithmeticException{
5           System.out.println("Inside check function");
6           throw new ArithmeticException("String");
7       }
8       public static void main(String[] args) {
9           try {
10              check();
11          }
12          catch(ArithmeticException e){
13          System.out.println("caught" + e);
14          }
15      }
16  }
```

OUTPUT

```
run:
Inside check function
caughtjava.lang.ArithmeticException: String
```

finally Keyword

A finally keyword is used to create a block of code that follows a try block. **A finally block of code is always executed whether an exception has occurred or not**.

Using a finally block lets one run any cleanup type statements that he/she would like to execute, no matter what happens in the protected code. A finally block appears at the end of catch block.

Example finally Block

In the example below, we are using finally block along with try block. This program throws an exception and due to exception, program terminates its execution but still the code written inside the finally block executed.

It is because of the nature of finally block that guarantees the execution of the code.

Example finally Block+

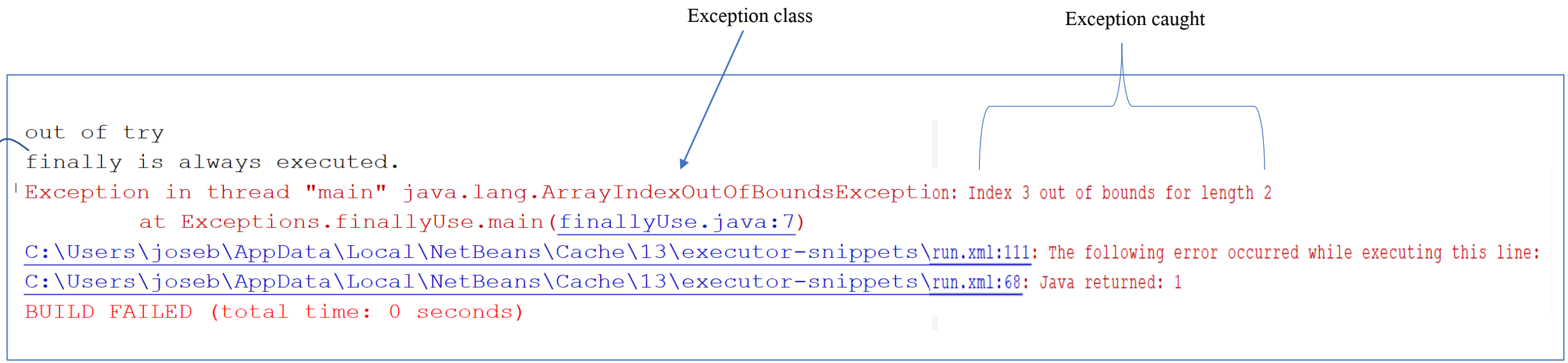
```
class finallyUse{
	public static void main(String[] args) {
		int a[] = new int[2];
		System.out.println("Out of try");
		try {
			System.out.println("Out of bound error "+ a[3]);
		}
		finally {
			System.out.println("finally is always executed.");
		}
	}
}
```

NOTE! catch block is not included, so we expect default exception handler to take action.

The code

```
1   package Exceptions;
2   public class finallyUse {
3       public static void main(String[] args) {
4           int a[] = new int[2];
5           System.out.println("out of try");
6           try {
7           System.out.println("Out of bound error "+ a[3]);
8           /* the above statement will throw
9           ArrayIndexOutOfBoundException */                }
10          finally {
11          System.out.println("finally is always executed.");
12          }
13  }}
```

Output



You can see in above example, even if exception is thrown by the program and it is not handled by catch block, still finally block gets executed.

Example : Finally Block with catch block

finally block executes in all the scenario whether exception is caught or not.

In previous example, we used finally where exception was not caught but here exception is caught and finally is used with **handler(catch block)**.

Finally Block with catch block++

Note the deployment of catch block in this program

```
class fiallyWithCatch{
      public static void main(String[] args){
            int a[] = new int[2];
            try {
            System.out.println("Access invalid element"+ a[3]);
            }
            catch(ArrayIndexOutOfBoundsException e){
                  System.out.println("Exception caught");
            }
            finally {
            System.out.println("finally is always executed.");
            }
      }
}
```

The code

```
1   package Exceptions;
2   public class fiallyWithCatch {
3       public static void main(String[] args){
4           int a[] = new int[2];
5           try {
6           System.out.println("Access invalid element"+ a[3]);
7           }
8           catch(ArrayIndexOutOfBoundsException e){
9           System.out.println("Exception caught");
10          }
11          finally {
12          System.out.println("finally is always executed.");
13          }
14      }
15  }
```

```
run:
Exception caught
finally is always executed.
```

User defined Exception subclass in Java

User defined Exception subclass in Java

User defined exceptions or custom exceptions are exception classes created by the programmer.

Java provides rich set of built-in exception classes like: ArithmeticException, IOException, NullPointerException etc. all are available in the java.lang package and used in exception handling.

These exceptions are already set to trigger on pre-defined conditions such as when you divide a number by zero it triggers ArithmeticException, (studytonigh.com,n.d).

User defined Exception subclass in Java

Apart from default Exception classes, Java allows us to create our own exception class to provide own exception implementation. These type of exceptions allows us to define how to deal with new challenges as far as exception handling is concern.

You can create your own exception *simply by extending java* ***Exception*** class. You can define a constructor for your Exception (not compulsory) and you can override the toString() function to display your customized message on catch. Lets see an example.

Example1: Custom Exception

In this example, we are creating an exception class MyException that extends the Java Exception class and

```
package Exceptions;
class MyException extends Exception {
    private int ex;
    MyException(int a){
    ex = a;
    }
    public String toString(){
    return "MyException[" + ex +"] is less than zero";
    }
}
```

Example1: Custom Exception-

MyException class code

```
2   package Exceptions;
3   public class MyException extends Exception {
        private int ex;
5       MyException(int a){
            ex = a;
7       }
        public String toString(){
9           return "MyException[" + ex +"] is less than zero";
10      }
11  }
```

Example1: Custom Exception-

MyException1(main) class

```
package Exceptions;
class MyException1{
        public static void main(String[] args) {
                try {
                sum(-54, 10);
                }
                catch(MyException me){
                System.out.println(me); //it calls the toString() method of user-defined
Exception
                }
        }
        static void sum(int a,int b) throws MyException {
                if(a<0){
                throw new MyException(a); //calling constructor of user-defined exception class
                }
                else{
                System.out.println(a+b);
                }
        }
}
```

MyException1(main) class code

```
1  package Exceptions;
2  public class MyException1 {
3      public static void main(String[] args) {
4          try {
5          sum(-56, 10);
6          }
7          catch(MyException me){
8          System.out.println(me);
9          }
10     }
11     static void sum(int a,int b) throws MyException {
12         if(a<0){
13         throw new MyException(a);
14         }
15         else{
16         System.out.println(a+b);
17         }
18     }
19 }
```

Output

When a < 0

```
run:
MyException[-56] is less than zero
BUILD SUCCESSFUL (total time: 0 seconds)
```

When a > 0

```
run:
14
BUILD SUCCESSFUL (total time: 0 seconds)
```

Note! Addition can not take place if operand a is less than zero

Example2: Custom Exception- ItemNotFoundExcep class

In this example we created a class ItemNotFoundExcep that extends the Exception class and helps to generate our own exception implementation when the item is not found.

```
package Exceptions;

class ItemNotFoundExcep extends Exception{
    public ItemNotFoundExcep(String s){
        super(s);
    }

}
```

Example2: Custom Exception- ItemNotFoundExcep1 class

In this example we created a class **ItemNotFoundExcep1** class that helps to run our own exception implementation in ItemNotFoundExcep.

```
class ItemNotFoundExcep1{
        static void find(int arr[], int item)throws ItemNotFoundExcep{
        boolean flag = false;
        for (int i = 0; i < arr.length; i++){
                if(item == arr[i]) flag = true;
                }
                        if(!flag){
                        throw new ItemNotFoundExcep("Item Not Found");
                }
                else {
                System.out.println("Item Found");
                }
        }
        public static void main(String[] args) {
                try {
                        find(new int[]{12,25,45}, 10);
                }
                catch(ItemNotFoundExcep i){
                        System.out.println(i);
                }
        } }
```

The code

```
1  package Exceptions;
2  public class ItemNotFoundExcep extends Exception{
3      public ItemNotFoundExcep(String s){
4          super(s);
5
6      }
7  }
```

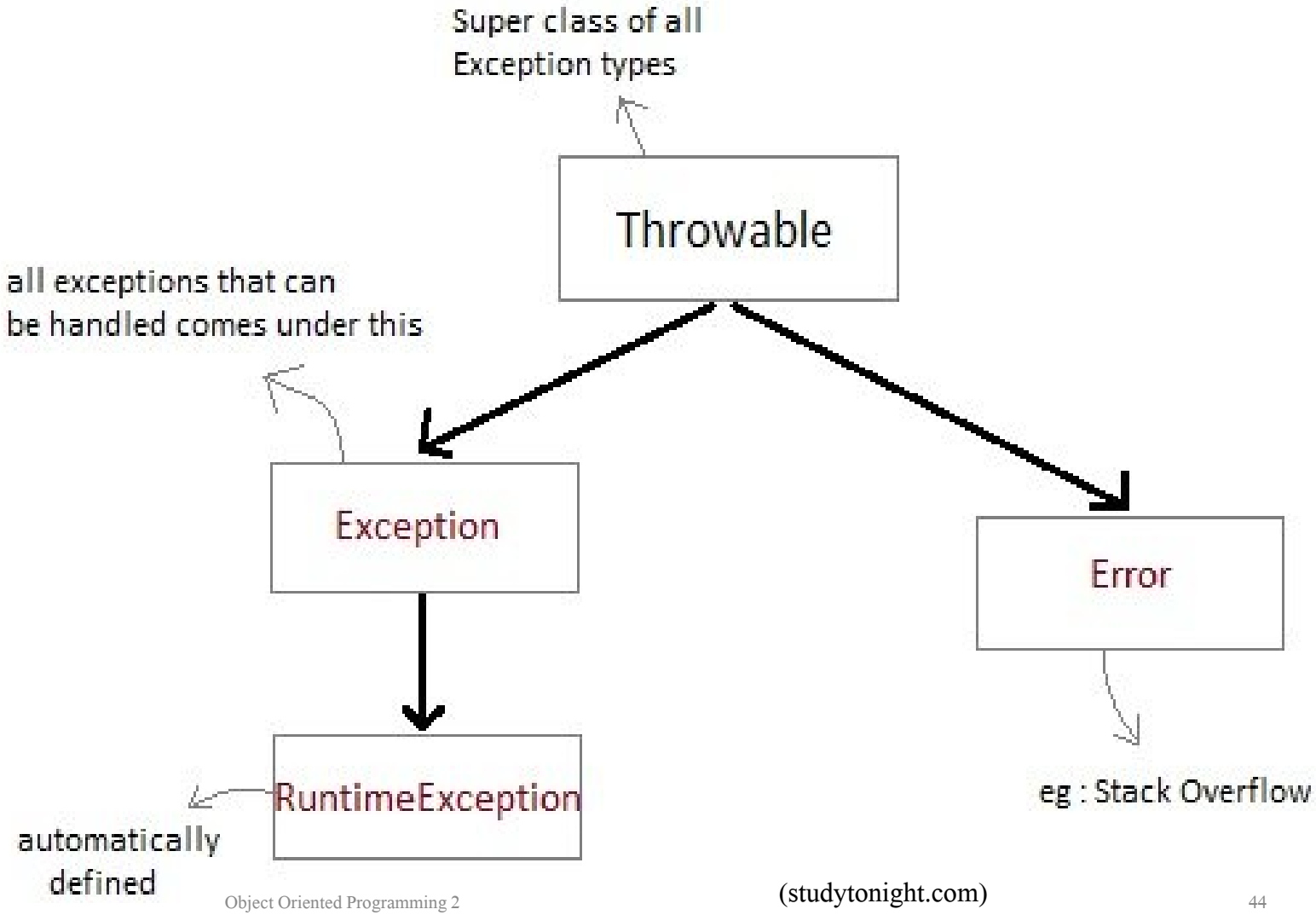
```
1  package Exceptions;
2  public class ItemNotFoundExcep1 {
3      static void find(int arr[],int item)
4                                      throws ItemNotFoundExcep{
5          boolean flag = false;
6          for (int i = 0; i < arr.length; i++){
7              if(item == arr[i]) flag = true;
8          }
9              if(!flag){
10                 throw new ItemNotFoundExcep("Item Not Found");
11             }
12             else {
13                 System.out.println("Item Found");
14             }
15         }
16         public static void main(String[] args) {
17             try { find(new int[]{12,25,45}, 10);
18             }
19             catch(ItemNotFoundExcep i){
20                 System.out.println(i);
21 } } }
```

Output

```
run:
Exceptions.ItemNotFoundExcep: Item Not Found
BUILD SUCCESSFUL (total time: 0 seconds)
```

Java Exception class Hierarchy

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



(studytonight.com)

Method Overriding with Exception Handling,

There are few things that one has to take note of when overriding a method with exception handling because method of **super class** may have exception declared. In this scenario, there can be possible two cases: *either method of super class can declare exception or not.*

If super class method declared exception then the method of sub class may declare same exception class, sub exception class or no exception but can not declare parent exception class.

Method Overriding with Exception Handling+

For example, if a method of super class declared ArithmeticException then method of subclass may declare ArithmeticException, its subclass or no exception but cannot declare its super/parent class like: Exception class.

In other scenario, if *super class method does not declare any exception*, then sub class overridden method cannot declare checked exception but it can declare unchecked exceptions. Lets see an example.

Method overriding in Subclass with Checked Exception supper class -Boss

Checked exception is the exception which is expected or known to occur at compile time hence they must be handled at compile time.

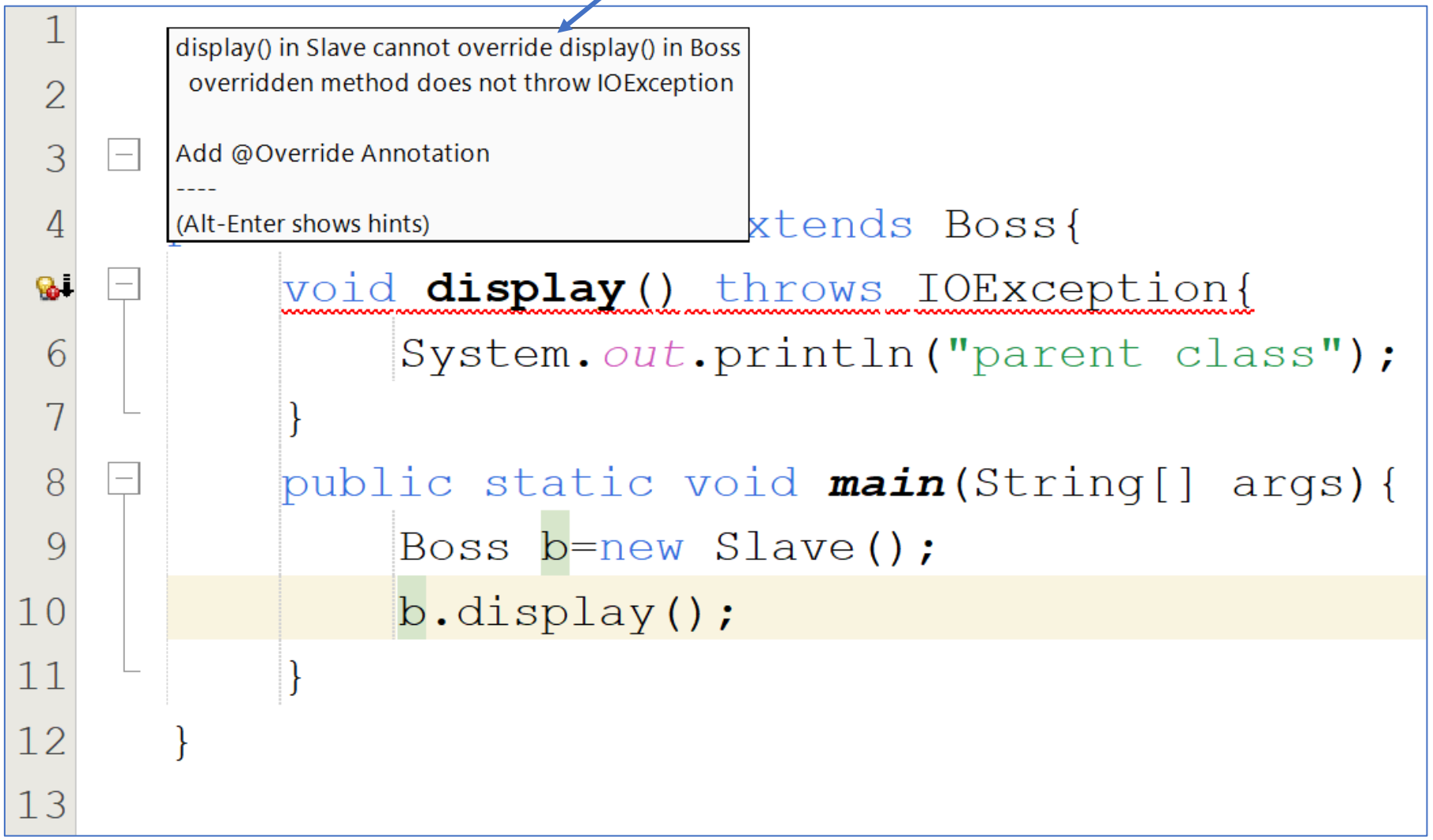
```
import java.io.*;
class Boss{
    void display(){
    System.out.println("parent class");
    }
}
```

Method overriding in Subclass with Checked Exception+ Sub Class-Slave

```
import java.io.*;
public class Slave extends Boss{
      void display() throws IOException{
      System.out.println("parent class");
      }
      public static void main(String[] args){
      Boss b=new Slave();
      b.display();
      }
}
```

In the example above, the method display() doesn't throw any exception when its declared/defined in the Boss class, hence its overridden version in the class Slave also cannot throw any checked exception. *If we try to do so, we* ***will get a compile time error***.

Method overriding in Subclass with Checked Exception+ Sub Class-Slave – Compile time error



Method overriding in Subclass with UnChecked Exception-Super class; Boss1

Unchecked Exceptions are the exception which extend the RuntimeExeption class and are thrown as a result of some runtime error.

```
import java.io.*;
class Boss1{
     void display(){
     System.out.println("parent class");
     }
}
```

Method overriding in Subclass with UnChecked Exception+ Sub Class-Slave1

```
import java.io.*;
public class Slave1 extends Boss1{
     void display() throws ArrayIndexOutOfBoundsException{
     System.out.println("child class");
     }
     public static void main(String[] args){
     Boss1 b=new Slave1();
     b.display();
     }
}
```

Because **ArrayIndexOutOfBoundsException** is a unchecked exception hence, overriden display() method can throw it.

Method overriding in Subclass with UnChecked Exception+ Sub Class-Slave1

```
1   package Exceptions;
2   import java.io.*;
3   public class Slave1 extends Boss1{
4       void display() throws ArrayIndexOutOfBoundsException{
5           System.out.println("child class");
6       }
7       public static void main(String[] args){
8           Boss1 b=new Slave1();
9           b.display();
10      }
11  }
```

```
run:
child class
```

More about Overriden Methods and Exceptions

If Super class method throws an exception, then Subclass overriden method can throw the same exception or no exception, but must not throw parent exception of the exception thrown by Super class method.

It means, if Super class method throws object of **NullPointerException** class, then Subclass method can either throw same exception, or can throw no exception, but it can never throw object of **Exception** class (parent of NullPointerException class), (Studytonight.com,n.d.).

Example of Subclass overriden method with same Exception-Boss2

Method of a sub class can declare same exception as declared in the super class. See the below example.

```
import java.io.*;
class Boss2{
     void display()throws Exception{
     System.out.println("Parent class");
     }
}
```

Example of Subclass overriden method with same Exception-Slave2

```
import java.io.*;
public class Slave2 extends Boss2{
	void display() throws Exception{
	System.out.println("Child class");
	}
	public static void main(String[] args){
	try{
		Boss2 b=new Slave2();
		b.display();
	}
	catch(Exception e){

	}
	}
}
```

Example of Subclass overriden method with same Exception-Slave2+

```
1   package Exceptions;
2   import java.io.*;
3   public class Slave2 extends Boss2{
4       void display() throws Exception{
5           System.out.println("Child class");
6       }
7       public static void main(String[] args){
8           try{
9               Boss2 b=new Slave2();
10              b.display();
11          }
12          catch(Exception e){
13          }
14      }
15  }
```

```
run:
Child class
```

Example of Subclass overriden method with no Exception-Boss3

It is optional to declare the exception in sub class during overriding. If method of super class declared an exception then it is upto the subclass to declare exception or not. See the below example.

```
import java.io.*;
class Boss3{
     void display()throws Exception{
     System.out.println("Parent class");
     }
}
```

Note! Boss3 class has display() method that throws Exception.

Example of Subclass overriden method with no Exception-Slave3 class

```
import java.io.*;
public class Slave3 extends Boss3{
      void display(){
      System.out.println("Child class");
      }
      public static void main(String[] args){
      try{
            Boss3 b=new Slave3();
            b.display();
      }
      catch(Exception e){
      }
      }
}
```

Note! No exception is being thrown by the Slave3 class display() method.

Example of Subclass overriden method with no Exception-Slave2 class+

```
1   package Exceptions;
2   public class Slave3 extends Boss3{
4           void display(){
5           System.out.println("Child class");
6           }
7           public static void main(String[] args){
8           try{
9                   Boss3 b=new Slave3();
10                  b.display();
11          }
12          catch(Exception e){
13          } }
    }
```

```
run:
Child class
```

Chained Exception in Java

Chained Exception in Java

Chained exceptions are exceptions are related to one another. i.e one exception describes cause of another exception.

For example, consider a situation in which a method throws an **ArithmeticException** because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero.

The method will throw only **ArithmeticException** to the caller. So the caller would not come to know about the actual cause of exception. Chained Exception is used in such type of situations.

Chained Exception in Java

Two new constructors and two new methods were added to **Throwable** class to support chained exception

1. **Throwable**(Throwable cause), In this first constructor, the paramter **cause** specifies the actual cause of exception
2. **Throwable**(String str, Throwable cause) this constructor allows us to add an exception description in string form with the actual cause of exception.

The following are the two methods added to **Throwable** class.

1. **getCause()** method returns the actual cause associated with current exception.
2. **initCause()** set an underlying cause(exception) with invoking exception, (Studytonigh.com, n.d.).

Example Chained Exception

In this example, ArithmeticException was thrown by the program but the real cause of exception was IOException.

We set the cause of exception using initCause() method.

```
import java.io.IOException;
public class ChainedException{
        public static void divide(int a, int b) {
        if(b == 0) {
        ArithmeticException ae = new ArithmeticException("Top layer");
        ae.initCause(new IOException("cause"));
        throw ae;
        }
        else{
                System.out.println(a/b);
        }
        }
        public static void main(String[] args){
        try {  divide(5, 0);
        }
        catch(ArithmeticException ae){
                System.out.println( "caught : " +ae);
                System.out.println("actual cause: "+ae.getCause());
        }
} }
```

Code with user input

```
1  package Exceptions;
2  import java.io.IOException;
3  import java.util.Scanner;
4  public class ChainedException{
5     static Scanner ob = new Scanner(System.in);
6      public static void divide(int a, int b) { if(b == 0) {
7          ArithmeticException ae =
8                  new ArithmeticException("Top layer");
9          ae.initCause(new IOException("cause"));
10         throw ae;
11         }
12         else{
13             System.out.println(a/b);
14         }
15     }
16     public static void main(String[] args){
17         System.out.println("Enter two values to divide ");
18         int    x = ob.nextInt();
19         int  y = ob.nextInt();
20         try {
21         divide(x, y);
22         }
23         catch(ArithmeticException ae){
24         System.out.println( "caught : " +ae);
           System.out.println("actual cause: "+ae.getCause());
26         }
27     }
28 }
```

Note the import of Scanner class on line 3, instantiation of the Scanner class on line 5, and user input instruction and storage from line 17 to Line 19.

Now this program will catch exceptions based on user input since divide() method is being provided with dynamic values x and y.

Code with user input-output

Output when value of x=20, and y = 2

```
run:
Enter two values to divide
20
2
10
```

Output when value of x=20, and y = 0

```
run:
Enter two values to divide
20
0
caught : java.lang.ArithmeticException: Top layer
actual cause: java.io.IOException: cause
```

Exception propagation

In Java, an exception is thrown from the top of the stack, if the exception is not caught it is put in the bottom of the stack, this process continues until it reaches to the bottom of the stack and caught.

It is known as exception propagation. By default, an unchecked exception is forwarded in the called chain.

Example1

```
class ExpPropagation{
	void a1(){
		int data = 30 / 0;
	}
	void a2(){
		a1();
	}
	void a3(){
	try {
		a2();
	}
	catch (Exception e){
		System.out.println(e);
	}
	}
	public static void main(String args[]) {
	ExpPropagation ob = new ExpPropagation();
		ob.a3();
	}
}
```

In this example, an exception occurred in method a1() which is called by method a2() and a2() is called by method a3().

Method a2() is enclosed in try block by a3() to provide the safe guard. We know exception will be thrown by method a1() but handled in method a3(). This is called exception propagation.

Example1-Output

```
1   package Exceptions;
2   class ExpPropagation{
3       void a1(){
            int data = 30 / 0;
5       }
6       void a2(){
            a1();
8       }
9       void a3(){
10          try {
11          a2(); }
12          catch (Exception e){
13              System.out.println(e);
14          } }
15      public static void main(String args[]) {
16          ExpPropagation ob = new ExpPropagation();
17          ob.a3();
18  } }
```

Output

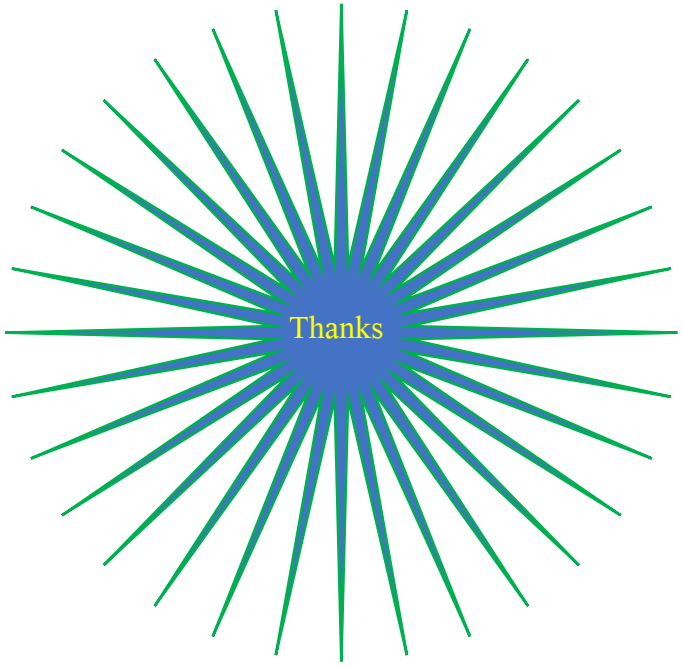
```
run:
java.lang.ArithmeticException: / by zero
```

Qn. Modify a1() method of the above program to allow User Input.

Summary

1. Java **try** with Resource Statement,
2. Java **throw**, **throws** and **finally** Keyword
3. Dealing with user defined Exception Classes
4. Method Overriding with Exception Handling,
5. Chained Exceptions(definition, exception propagation)

Thank you for Listening



References

Chained exception in Java. Studytonight.com. (n.d.). Retrieved September 24, 2022, from https://www.studytonight.com/java/chained-exception-in-java.php

User defined exception subclass in Java. Studytonight.com. (n.d.). Retrieved September 18, 2022, from https://www.studytonight.com/java/create-your-own-exception.php

Java method overriding with exception handling. Studytonight.com. (n.d.). Retrieved September 17, 2022, from https://www.studytonight.com/java/methodoverriding-with-exception-handling.php

Java exception handling. Studytonight.com. (n.d.). Retrieved September 12, 2022, from https://www.studytonight.com/java/exception-handling.php

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