Marwadi University

Unit – 5
Exception Handling

Prepared By
Prof. Ravikumar Natarajan
Assistant Professor, CE Dept.

Contents

Error vs Exception

- In java, both Errors and Exceptions are the subclasses of **java.lang.Throwable** class.

- Error refers to an illegal operation performed by the user which results in the abnormal working of the program.
- Recovering from error is difficult.
- **Ex: java.lang.OutOfMemoryError**

- Exceptions in java refer to an unwanted or unexpected event, which occurs during the execution of a program i.e at run time & compile time, that disrupts the normal flow of the program's instructions.
- Recovering from exception is possible by using try, catch, or throwing exceptions.
- **Ex: Checked and Unchecked Exceptions**

Exception Handling

- Exception handling enables a program to deal with exceptional situations and continue its normal execution.

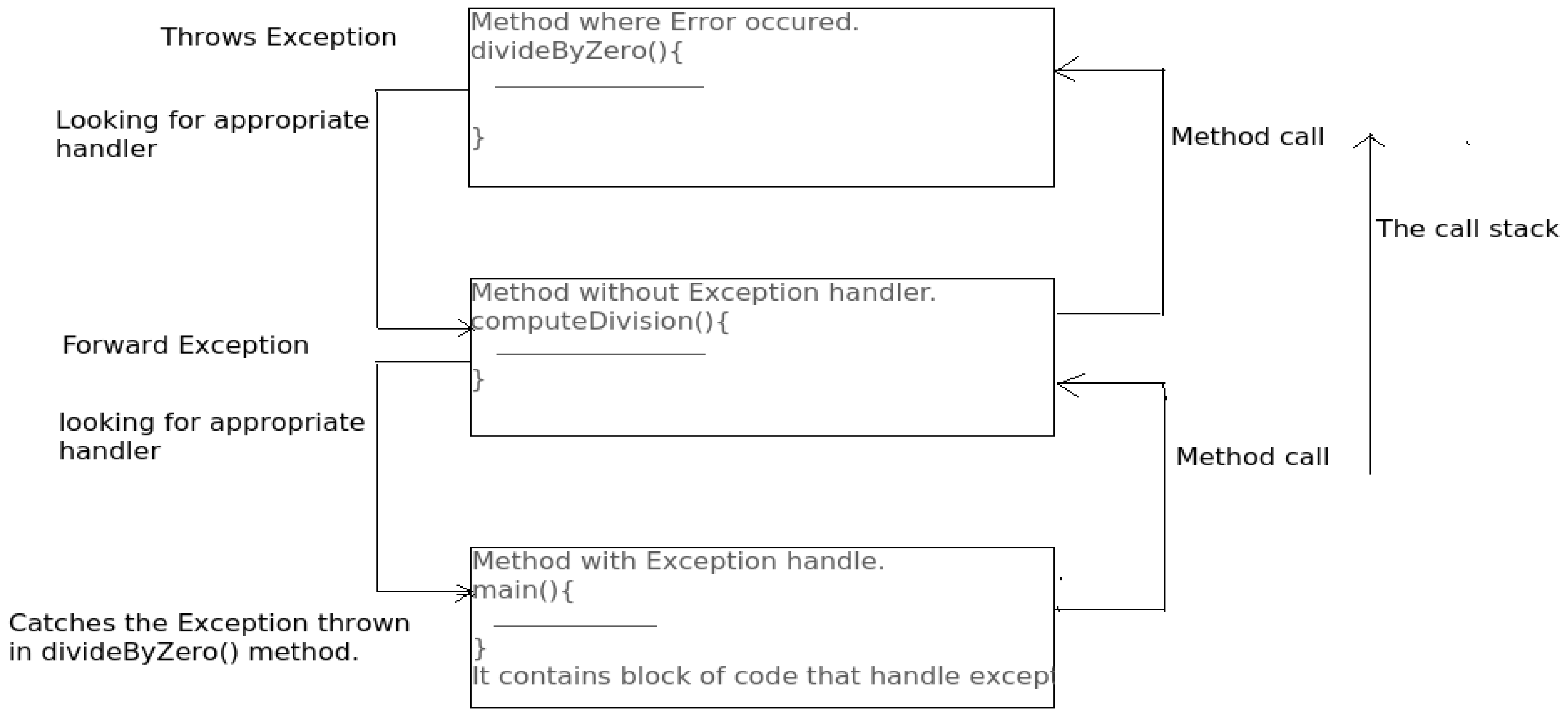
- Handles runtime & compile time errors.

- Ex. You divide a number with ‘0’. Division by 0 exception will occur and program will be terminated.
- If you can handle this kind of exceptions then your program will continue its normal execution.

What Happens During Exception?

- Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at **run time,** that disrupts the normal flow of the program's instructions.
- 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.

What is Call Stack?


Throws Exception
Looking for appropriate handler
Forward Exception
looking for appropriate handler
Catches the Exception thrown in divideByZero() method.
Method where Error occured.
divideByZero(){
}
Method without Exception handler.
computeDivision(){
}
Method with Exception handle.
main(){
}
It contains block of code that handle except
Method call
Method call
The call stack

The call stack and searching the call stack for exception handler.

Creating the Exception Object and handling it in the run-time system is called **throwing an Exception**. There might be a list of the methods that had been called to get to the method where an exception occurred. This ordered list of the methods is called **Call Stack.**

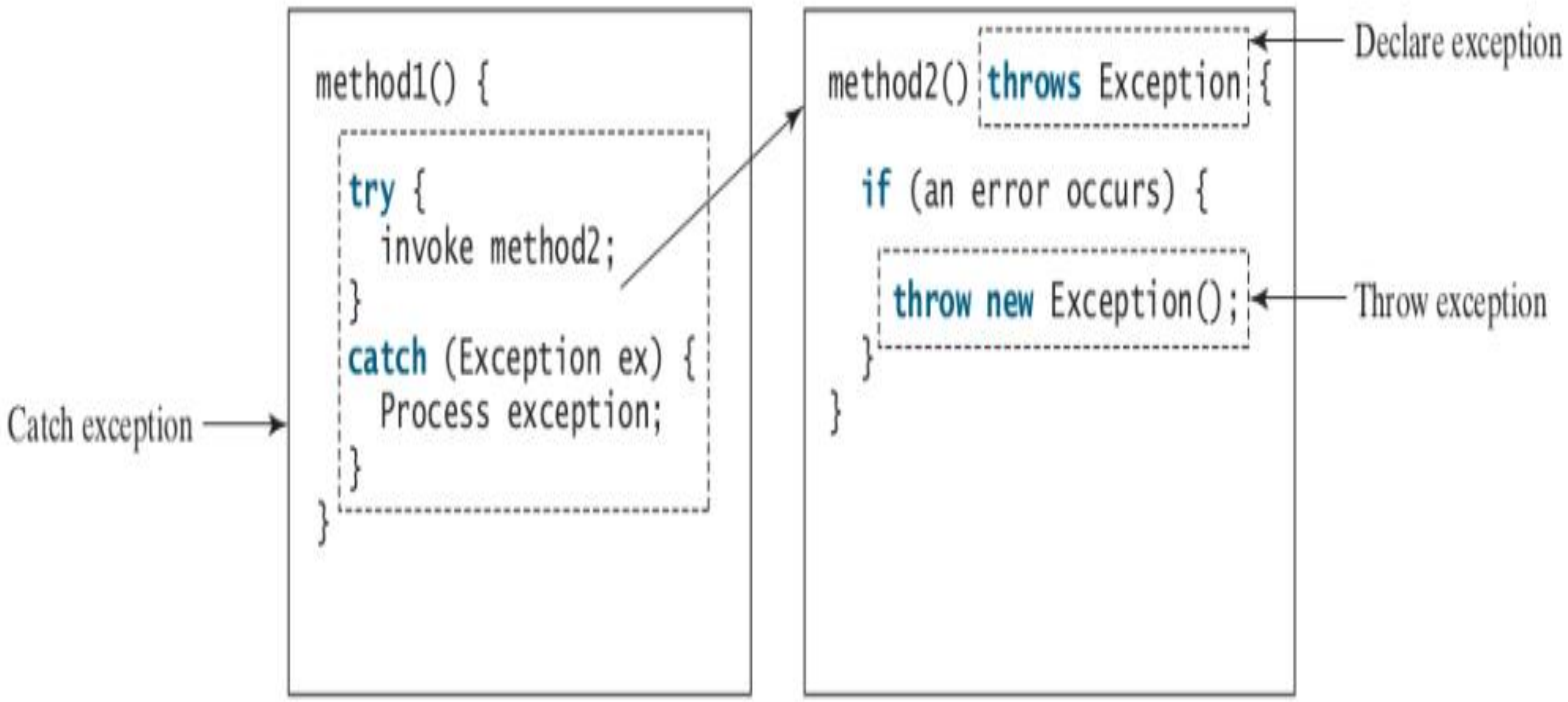
Exception-Handling Overview

Java’s exception-handling model is based on three operations:

1. Declaring an exception,
2. Throwing an exception, and
3. Catching an exception.

Exceptions are thrown from a method. The caller of the method can catch and handle the exception.

Exception-Handling Overview


```
method1() {

  try {
    invoke method2;
  }
  catch (Exception ex) {
    Process exception;
  }
}
```

```
method2() throws Exception {

  if (an error occurs) {

    throw new Exception();
  }
}
```

Exception Types

Three types,

1. Checked Exception (Compile time)

These exceptions checked by the code itself. Using try-catch or throws i.e compiler will check these exceptions. From java.lang.Exception class.

Ex: IOException

2. Unchecked Exception (Run time)

These exceptions are not checked by compiler. JVM will check these exceptions. From java.lang.RuntimeException class.

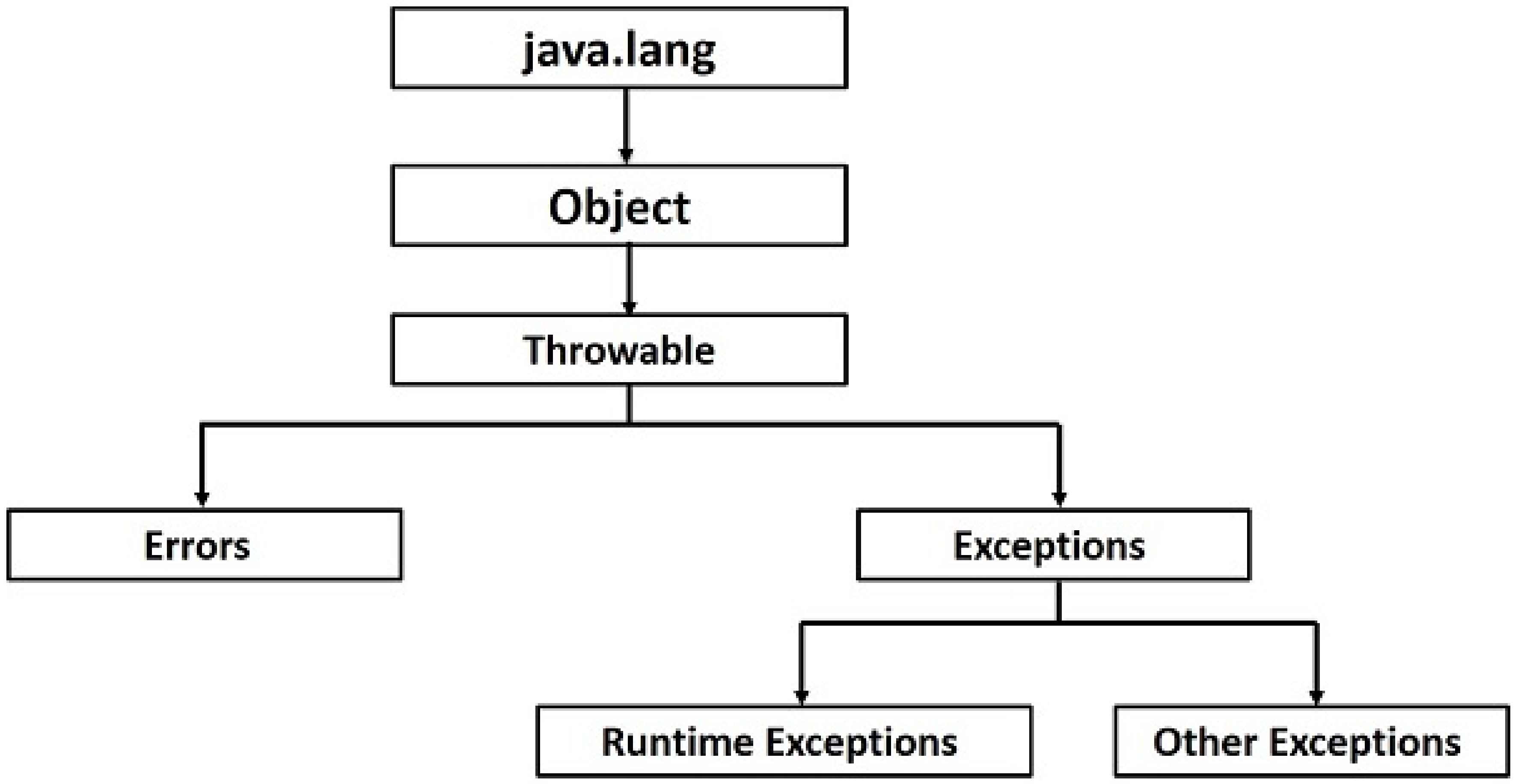
Ex: ArrayIndexOutOfBounds, RunTimeException.

3. System Errors

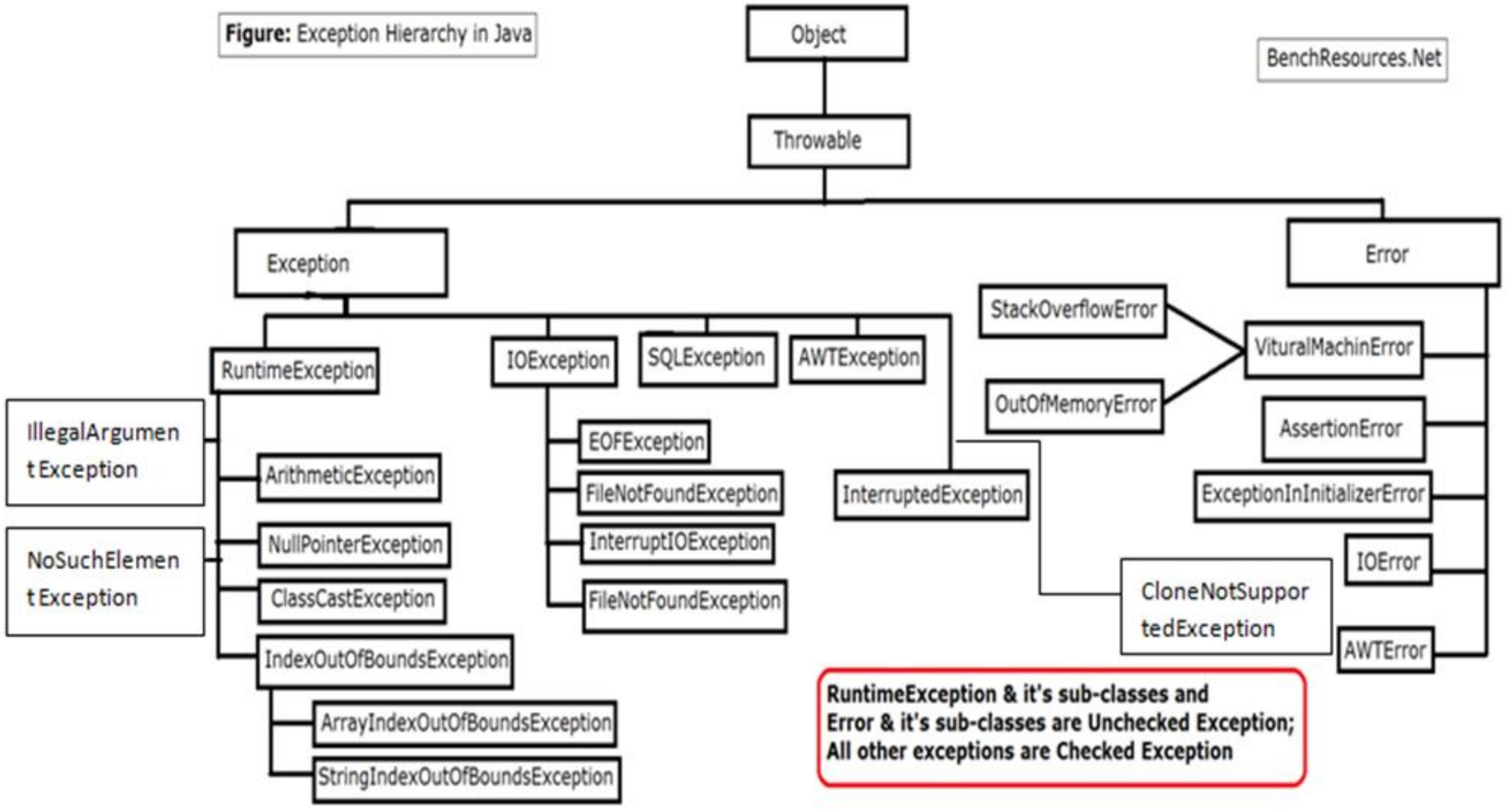
System errors are thrown by the JVM and are represented in the Error class. The Error class describes **internal** system errors, though such errors rarely occur.

Ex: LinkageError, VirtualMachineError

Exception Types

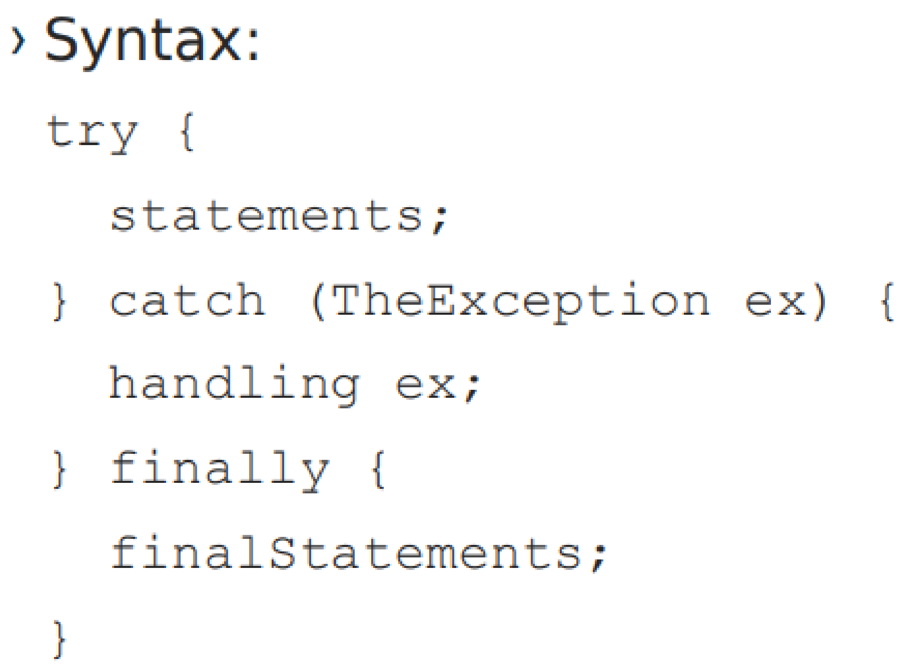


Exception Types



The finally Clause

- The finally clause is always executed regardless whether an exception occurred or not.
- You may want some code to be executed regardless of whether an exception occurs or is caught.
- Java has a finally clause that can be used to accomplish this objective.



› Syntax:

```
try {
  statements;
} catch (TheException ex) {
  handling ex;
} finally {
  finalStatements;
}
```

Throwing and Catching Exceptions

Five keywords to handle exception,

1. Try – to monitor exception | to try critical block
2. Catch – handles specific exception with try block
3. Finally – code executed even exception may or may not occur. Denotes end of the program. | optional to use
4. Throw – used to throw specific exception
5. Throws - used to throw specific exception by a particular method.

Throwing and Catching Exceptions

try-catch-finally keyword

```
class myExeption
{           public static void main(String s[]){
                        int i=5, j=0;
                        System.out.println("Try started");
                        try
                        {
                                    int temp = i/j;
                                    System.out.println("Inside try");
                        }
                        catch(Exception e)
                        {
                                    System.out.println("Inside catch");
                                    System.out.println("Divide by 0");
                        }
                        finally
                        {
                                    System.out.println("Finally block");
                        }           }   }
```

Output:
```
try started
Inside catch
Divide by 0
Finally block
```

Throwing and Catching Exceptions

Using Multiple Catch Clauses

To catch different types of exceptions multiple catch clause can be used.

Example:

```
public class TestMultipleCatchBlock{
public static void main(String args[]){
                    try{
                              int a[]=new int[5];
                              a[5]=30/0;
                       }
                    catch(ArithmeticException e){
                              System.out.println("task1 is completed");}
                    catch(ArrayIndexOutOfBoundsException e){
                              System.out.println("task 2 completed");}
                     catch(Exception e){
                              System.out.println("common task completed");}
                    System.out.println("rest of the code...");
 } }
```

```
Output:task1 completed
    rest of the code...
```

Throwing and Catching Exceptions

Try block can be nested

A try, catch or finally block can contain another set of try catch and finally sequence.

```
class Excep6{
    public static void main(String args[]){
     try{
                         try{
                                     System.out.println("going to divide");
                                      int b =39/0;
                         }catch(ArithmeticException e)    {System.out.println(e);}

                          try{
                                      int a[]=new int[5];
                                     a[5]=4;
                         }catch(ArrayIndexOutOfBoundsException e) {System.out.println(e);}
                         System.out.println("other statement");
     }catch(Exception e)
{System.out.println("handeled");}  System.out.println("normal flow..");       }
```

Output:

```
going to divide
Java.lang. ArithmeticException: /by zero
Java.lang. ArrayIndexOutOfBoundsException: 5
Other statement
Normal flow..
```

Methods to Print the Exception Information

1. printStackTrace()
2. toString()
3. getMessage()

```
import java.io.*;
class Main {
        public static void main (String[] args) {
        int a=5;
        int b=0;
                try{
                System.out.println(a/b);
                }
        catch(ArithmeticException e){
                e.printStackTrace();
                System.out.println(e.toString());
                System.out.println(e.getMessage());
        }
        }
}
```

Activity

- Go to www.menti.com
- 36983345

Throwing and Catching Exceptions

Using throw #UserDefined Exception **#Explicit**

```
public class Main
{
  void checkAge(int age)
          {  if(age<18)
              throw new ArithmeticException("Not Eligible for voting");  //inside Method UserDefined
          else
                    System.out.println("Eligible for voting");
          }
  public static void main(String args[])
          {
           Main obj = new Main();
          obj.checkAge(13);
          System.out.println("End Of Program");
          }
}
```

Output:

```
Exception in thread "main"
java.lang.ArithmeticException:
Not Eligible for voting
at Example1.checkAge(Example1.java:4)
at Example1.main(Example1.java:10)
```

Throwing and Catching Exceptions

Example using throws

```
public class Example1
{
   int division(int a, int b) throws ArithmeticException  //Method signature, Supports Multiple Exception
            {  int t = a/b;
                return t;
            }
   public static void main(String args[])
            {
            Example1 obj = new Example1();
                        try{
                                System.out.println(obj.division(15,0));
                           }
                        catch(ArithmeticException e)
                          {
                                System.out.println("You shouldn't divide number by zero");
                           }
            }
}
```

Output:

```
You shouldn't divide number by zero
```

Rethrowing Exception

- Sometimes we may need to rethrow an exception in Java. If a catch block cannot handle the particular exception it has caught, we can rethrow the exception. The rethrow expression causes the originally thrown object to be rethrown.
- Java allows an exception handler to rethrow the exception if the handler cannot process the exception or simply wants to let its caller be notified of the exception.

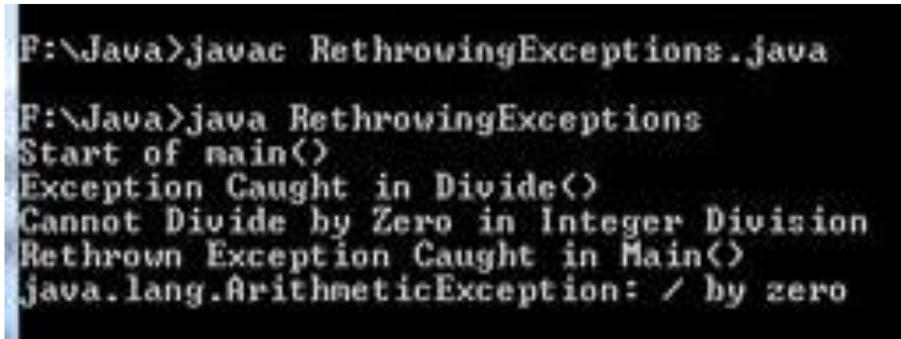
- The syntax for rethrowing an exception may look like this:

```
try {
 statements;
} catch (TheException ex) {
 perform operations before exits;
 throw ex;
}
```

Rethrowing Exception

```
public class RethrowingExceptions
 {   static void divide() {
        int x,y,z;
        try  {
           x = 6 ;
           y = 0 ;
           z = x/y ;
       System.out.println(x + "/"+ y +" = " + z);
         }
        catch(ArithmeticException e)
         {
          System.out.println("Exception Caught
in Divide()");
          System.out.println("Cannot Divide by
Zero in Integer Division");
          throw e; // Rethrows an exception
         }
     }
```

```
 public static void main(String[] args)
 {
        System.out.println("Start of main()");
        try
       {
          divide();
        }
        catch(ArithmeticException e)
       {
           System.out.println("Rethrown
Exception Caught in Main()");
           System.out.println(e);
       }
  }
}
```



```
F:\Java>javac RethrowingExceptions.java

F:\Java>java RethrowingExceptions
Start of main()
Exception Caught in Divide()
Cannot Divide by Zero in Integer Division
Rethrown Exception Caught in Main()
java.lang.ArithmeticException: / by zero
```

Chained Exception

Throwing an exception along with another exception forms a chained exception.

Example

```
// the catch block of method1
catch (Exception ex) { ex.printStackTrace(); }

// the catch block of method2
catch (Exception ex) {
        throw new Exception("New info from method1", ex);
}
```

Chained Exception

The Throwable class has methods which support exception chaining −

Method	Description
getCause()	Returns the original cause of the exception
initCause(Throwable cause)	Sets the cause for invoking the exception

```
public class Example {
  public static void main(String[] args) {
    try {
      // creating an exception
      ArithmeticException e = new ArithmeticException("Apparent cause");
      // set the cause of an exception
      e.initCause(new NullPointerException("Actual cause"));
      // throwing the exception
      throw e;
    } catch(ArithmeticException e) {
      // Getting the actual cause of the exception
      System.out.println(e.getCause());      }    } }
```

Defining Custom Exception Classes

- You can define a custom exception class by extending the java.lang.Exception class.

- You can use exception classes provided by Java, whenever it is appropriate.
- If you run into a problem that cannot be adequately described by the predefined exception classes, you can create your own exception class.
- Custom Exception Class must be derived from Exception or from a subclass of Exception, such as IOException.

Defining Custom Exception Classes

```
class CustomException extends Exception {
   String message;
   CustomException(String str) {
      message = str;
   }
   public String toString() {
      return ("Custom Exception Occurred : " + message);
   }
}
public class MainException {
   public static void main(String args[]) {
      try {
         throw new CustomException(“Hello!! this my message”);
      } catch(CustomException e) {
         System.out.println(e);       }    } }
```

Questions?

- **When to use throws throw VS try-catch in Java?**
- The "**throws**" keyword is used to declare the exception with the method signature.
- The **throw** keyword is used to explicitly throw the exception.
- The **try** statement defines the code block to run (to try).
- The **catch** statement defines a code block to handle any error.

- **What do you know about finally block?**
- The **finally** statement defines a code block to run regardless of the result.
- **Which can handle multiple exceptions throw or throws?**
- **Throws**

Summary

- Exception Handling Overview
- Types of Exception
- Using Try
- Catch and Finally Clauses
- Multiple Catch Clauses
- Throw and Throws Keyword
- Custom Exception Class

Next

- Multithreading
- Thread Model
- Creating Threads
- Thread Priorities
- Synchronization
- Inter-thread Communication

END OF UNIT - 5