



# EXCEPTION

CSI 211: OBJECT ORIENTED PROGRAMMING

Tanjina Helaly

# WHAT IS EXCEPTION

- An *exception* is an abnormal condition that arises in a code sequence at run time.
- In other words, an exception is a runtime error.
- In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on.



# WHAT IS EXCEPTION

- Definition: An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- Following are some scenarios where an exception occurs.
  - A user has entered an invalid data.
  - A file that needs to be opened cannot be found.
  - A network connection has been lost in the middle of communications or the JVM has run out of memory.



# HOW DOES IT OCCUR?

- **a) Implicitly by some error condition – (system generated exception)**

```
class ImplicitlyRaisedException{  
    public static void main( String[] arguments ){  
        int students[] = new int[5];  
        students[ 10 ] = 1; // Exception occurs here  
    }  
}
```

- **b) Explicitly by the Programmer**

```
class ExplicitlyRaisedException{  
    public static void main( String[] arguments ){  
        throw new ArrayIndexOutOfBoundsException();  
    }  
}
```



## FEW MORE EXAMPLES

Example Method call	exception
<code>scan.nextInt()</code> – for any nonInteger input	<code>InputMismatchException</code>
<code>Integer.parseInt("abc")</code>	<code>NumberFormatException</code>
<code>2/0</code>	<code>ArithmeticException</code>
<code>new FileReader("C:\\temp.txt")</code> Or <code>new FileWriter("C:\\temp.txt")</code>	<code>FileNotFoundException</code>
<code>obj.read()</code> or <code>obj.write()</code> - obj is any IO related object	<code>IOException</code>



# EXCEPTION KEYWORDS

- try
- catch
- finally
- throw
- throws



# HOW TO HANDLE EXCEPTION

- Using try-catch-finally block
  - 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.
  - Programmer code can catch this exception (using **catch**) and handle it in some rational manner.
  - Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.



# FLOW CONTROL IN TRY-CATCH BLOCKS

- When you call a risky method, one of two things can happen.
  - The risky method either succeeds, and the **try block completes**,
  - or the risky method **throws an exception** back to your calling method.
- If the try block fails (throws an exception),
  - flow control **immediately** moves to the **catch** block without executing the rest of the code in try block..
  - When the **catch** block **completes**, the **finally** block runs.
  - When the **finally** block **completes**, the rest of the method continues on.





# FLOW CONTROL IN TRY-CATCH BLOCKS

- If the try block succeeds (no exception),
  - flow control **skips over the catch** block and **moves to the finally block**.
  - When the finally block completes, the rest of the method continues on.
- If the try or catch block has a **return** statement, finally will still run
  - Flow jumps to the **finally**, then **back to the return**.
- If the try or catch block has a **throw** statement and it is not handled, finally will still run
  - Flow jumps to the **finally**, then **back to the throw**.



# HOW TO HANDLE EXCEPTION

- This is the general form of an exception-handling block:

```
try {  
    // block of code to monitor for errors  
}  
catch (ExceptionType1 exOb) {  
    // exception handler for ExceptionType1  
}  
catch (ExceptionType2 exOb) {  
    // exception handler for ExceptionType2  
}  
// ...  
finally {  
    // block of code to be executed after try block ends  
}
```

- Here, *ExceptionType* is the type of exception that has occurred.



# EXAMPLE

```
class TestException {  
    public static void main(String args[]) {  
        int d, a;  
        try { // monitor a block of code.  
            d = 0;  
            a = 42 / d;  
            System.out.println("This will not be printed.");  
        } catch (ArithmeticException e) { // catch divide-by-zero error  
            System.out.println("Exception Message: "+e.getMessage());  
            System.out.println("Division by zero.");  
        }  
        System.out.println("After catch statement.");  
    }  
}
```

- This program generates the following output:

Exception Message: / by zero

Division by zero.

After catch statement.



# EXAMPLE – MULTIPLE CATCHES

```
import java.util.*;
public class TestException {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        boolean successful = false;
        while(!successful){
            System.out.println("Enter 2 integers.");
            try{
                int a = scan.nextInt();
                int b = Integer.parseInt(scan.nextLine().trim());
                int c = a/b;
                System.out.println("Result: " + c);
                successful = true;
            }
            catch(ArithmeticException e){
                System.out.println("Can not divide by 0.");
            }
            catch(InputMismatchException e){
                System.out.println("Need 2 numbers for division.");
                if (scan.hasNextLine())
                    scan.nextLine();
            }
            catch(NumberFormatException e){
                System.out.println("Need 2 numbers for division.");
            }
        }
        scan.close();
    }
}
```

```
<terminated> TestException [Java Application] C:\I
Enter 2 integers.
2 e
Need 2 numbers for division.
Enter 2 integers.
e 4
Need 2 numbers for division.
Enter 2 integers.
4 0
Can not divide by 0.
Enter 2 integers.
4 2
Result: 2
```



# THROWS

- If a method is
  - capable of causing an exception that it does not handle,
  - it must specify this behavior so that callers of the method can guard themselves against that exception.
- This is done by including a **throws** clause in the method's declaration.
- A **throws** clause lists all types of exceptions that a method might throw.
- This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses.



# THROW VS. THROWS

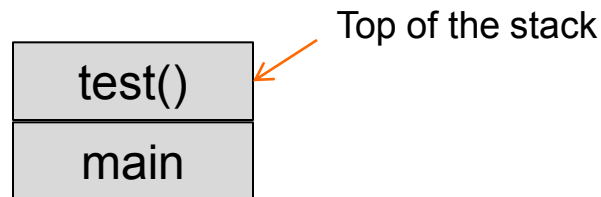
- System-generated exceptions are automatically thrown by the Java runtime system. To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- Example

```
Class TestException{  
    public void throwException() throws Exception{  
        throw new Exception();  
    }  
    public void throwSystemException() throws InterruptedException {  
        Thread.sleep(100);  
    }  
}
```



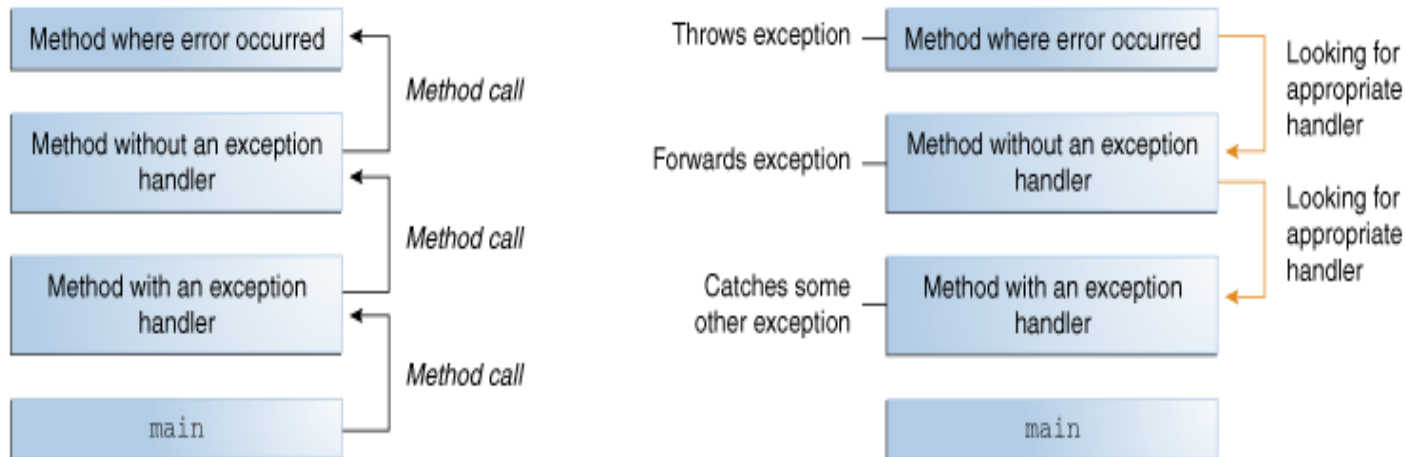
# METHODS ARE STACKED

- When you call a method, the method lands on the top of a call stack.
- The method at the *top of the stack is always* the currently-running method for that stack
- A method stays on the stack until the method hits its closing curly brace (which means the method's done).
- If method *main()* calls method *test()*, method *test()* is stacked on top of method *main()*



# METHOD STACK AND EXCEPTION

- Exception must be handled in one of the method in method stack.
- Once an exception is handled, the program continues normal execution.
- If you handle the same exception in multiple level only the closet one will be used to handle the exception.





# METHOD STACK AND EXCEPTION

```
public class TestException {  
    public static void main(String[] args){  
        int input = Integer.parseInt(args[0]);  
        testSqrt(input);  
    }  
    public static void testSqrt(int s){  
        try{  
            System.out.println(sqr(s));  
        }catch(Exception e) {  
            System.out.println(e.getMessage());  
            e.printStackTrace();}  
        }  
    public static int sqr(int a) throws Exception{  
        if (a < 0)  
            throw new Exception("can't be less than 0");  
        return a*a;  
    }  
    public static int callSqr(int a) throws Exception{  
        return sqr(a);  
    }  
}
```

## ○ Output:

```
<terminated> TestException [Java Application] C:\Program Files\Java\jdk1.8.0_31\bin\javaw.exe (Nc  
can't be less than 0  
java.lang.Exception: can't be less than 0  
    at testexception.TestException.sqr(TestException.java:63)  
    at testexception.TestException.testSqrt(TestException.java:76)  
    at testexception.TestException.main(TestException.java:56)
```



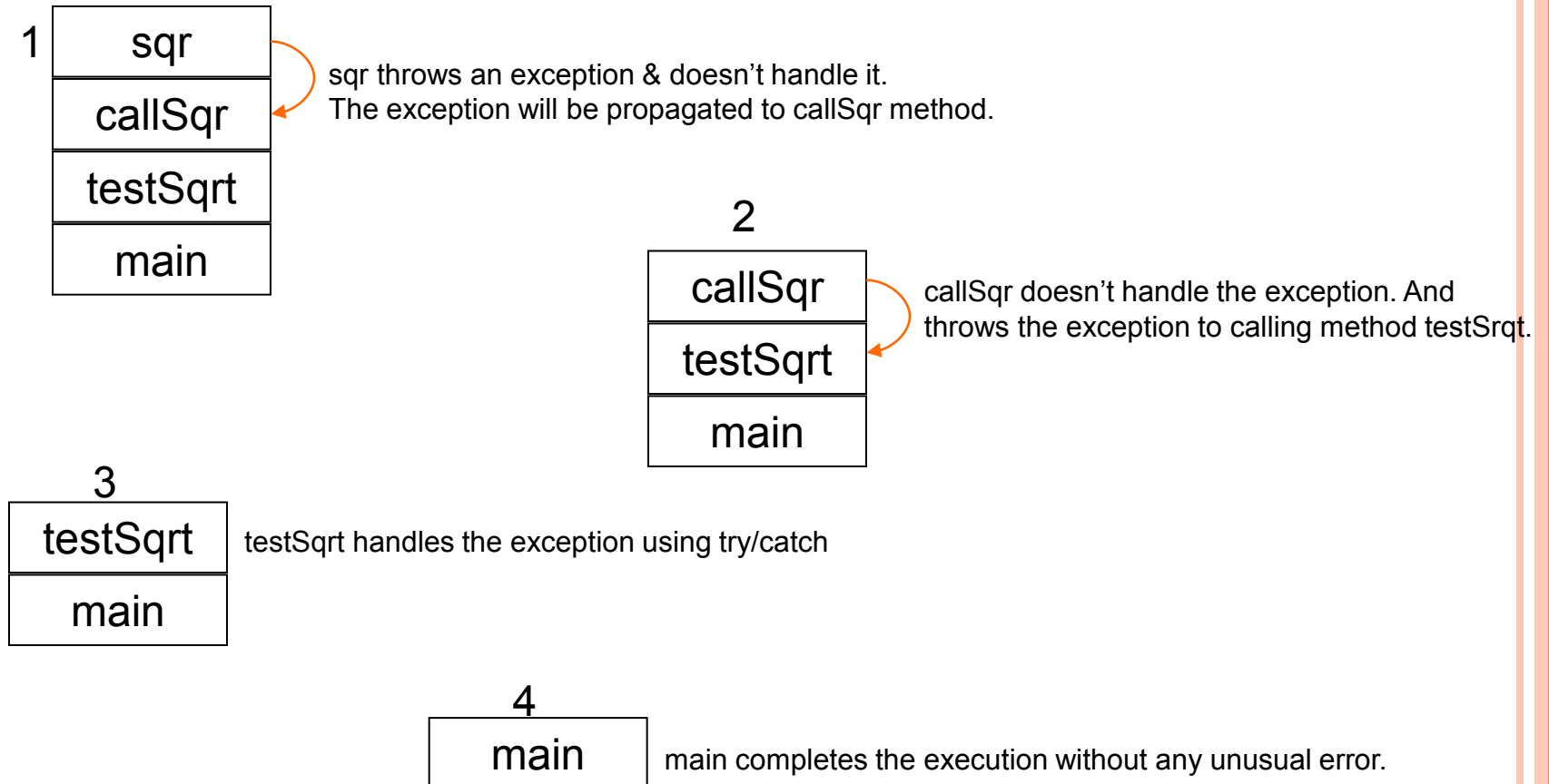
# METHOD STACK AND EXCEPTION

```
public class TestException {  
    public static void main(String[] args){  
        int input = Integer.parseInt(args[0]);  
        testSqrt(input);  
    }  
    public static void testSqrt(int s){  
        try{  
            System.out.println(sqr(s));  
        }catch(Exception e) {  
            System.out.println(e.getMessage());  
        }  
    }  
    public static int sqr(int a) throws Exception{  
        if (a < 0)  
            throw new Exception("can't be less than 0");  
        return a*a;  
    }  
  
    public static int callSqr(int a) throws Exception{  
        return sqr(a);  
    }  
}
```

Stack	
sqr	
callSqr	
testSqr	
main	



# METHOD STACK AND EXCEPTION



# EXAMPLE - NESTED TRY CATCH

- If you handle the same exception in multiple level only the closet one will be used to handle the exception.

```
public class multilevel {  
    public static void main(String[] args) {  
        int[] course = new int[10];  
        try{  
            System.out.println("Outer try");  
            try{  
                System.out.println( "Start Change" );  
                course[ 10 ] = 1;  
                System.out.println( "End Change" );  
            } catch(ArrayIndexOutOfBoundsException e) {  
                System.out.println( "Inner Catch: " + e.getMessage());  
            }  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println( "Outer Catch: " + e.getMessage());  
        }  
    }  
}
```

```
<terminated> multilevel [Java A  
Outer try  
Start Change  
Inner Catch: 10  
|
```



# EXAMPLE - NESTED TRY CATCH

- If an exception is not handled in inner level it can be handled by outer level.

```
public class multilevel {  
    public static void main(String[] args) {  
        int[] course = new int[10];  
        try{  
            System.out.println("Outer try");  
            try{  
                System.out.println( "Start Change" );  
                course[ 10 ] = 1;  
                System.out.println( "End Change" );  
            } catch(NumberFormatException e) {  
                System.out.println( "Inner Catch: " + e.getMessage());  
            }  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println( "Outer Catch: " + e.getMessage());  
        }  
    }  
}
```

```
<terminated> multilevel [Java Applicatio  
Outer try  
Start Change  
Outer Catch: 10  
|
```



# EXAMPLE-THROWING A DIFFERENT EXCEPTION

- It is possible to throw a different exception after catching an exception.
- If the try or catch block has a **throw** statement and it is not handled, finally will still run
  - Flow jumps to the **finally**, then **back to the throw**

```
public class TestFinally {  
    public static void main(String[] args){  
        try{  
            test();  
        }catch(Exception e){  
            System.out.println("Catch from main: "+ e.getMessage());  
        }  
    }  
    public static void test(){  
        try{  
            int c = 4/0; // system generated exception  
        } catch(ArithmeticException e){  
            System.out.println("Catch from test: "+ e.getMessage());  
            throw new IllegalArgumentException("throwing another exception");  
        } finally{  
            System.out.println("Finally from test method.");  
        }  
    }  
}
```

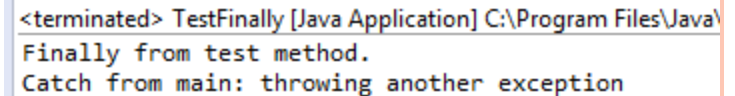
```
<terminated> TestFinally [Java Application] C:\Program Files\Ja  
Catch from test: / by zero  
Finally from test method.  
Catch from main: throwing another exception
```



# EXAMPLE-THROWING A DIFFERENT EXCEPTION

- It is possible to throw a different exception after catching an exception.
- If the try or catch block has a **throw** statement and it is not handled, finally will still run
  - Flow jumps to the **finally**, then **back to the throw**

```
public class TestFinally {  
    public static void main(String[] args){  
        try{  
            test();  
        }catch(Exception e){  
            System.out.println("Catch from main: "+ e.getMessage());  
        }  
    }  
    public static void test(){  
        try{  
            throw new IllegalArgumentException("throwing another exception");  
        } catch(ArithmeticException e){  
            System.out.println("Catch from test: "+ e.getMessage());  
        } finally{  
            System.out.println("Finally from test method.");  
        }  
    }  
}
```



```
<terminated> TestFinally [Java Application] C:\Program Files\Java\  
Finally from test method.  
Catch from main: throwing another exception
```



# EXAMPLE-THROW AND CATCH EXCEPTION IN SAME BLOCK

```
public class TestFinally {  
    public static void main(String[] args){  
        try{  
            test();  
        }catch(Exception e){  
            System.out.println("Catch from main: "+ e.getMessage());  
        }  
    }  
  
    public static void test(){  
        try{  
            throw new ArithmeticException("throwing an exception");  
        } catch(ArithmeticException e){  
            System.out.println("Catch from test: "+ e.getMessage());  
        } finally{  
            System.out.println("Finally from test method.");  
        }  
    }  
}
```

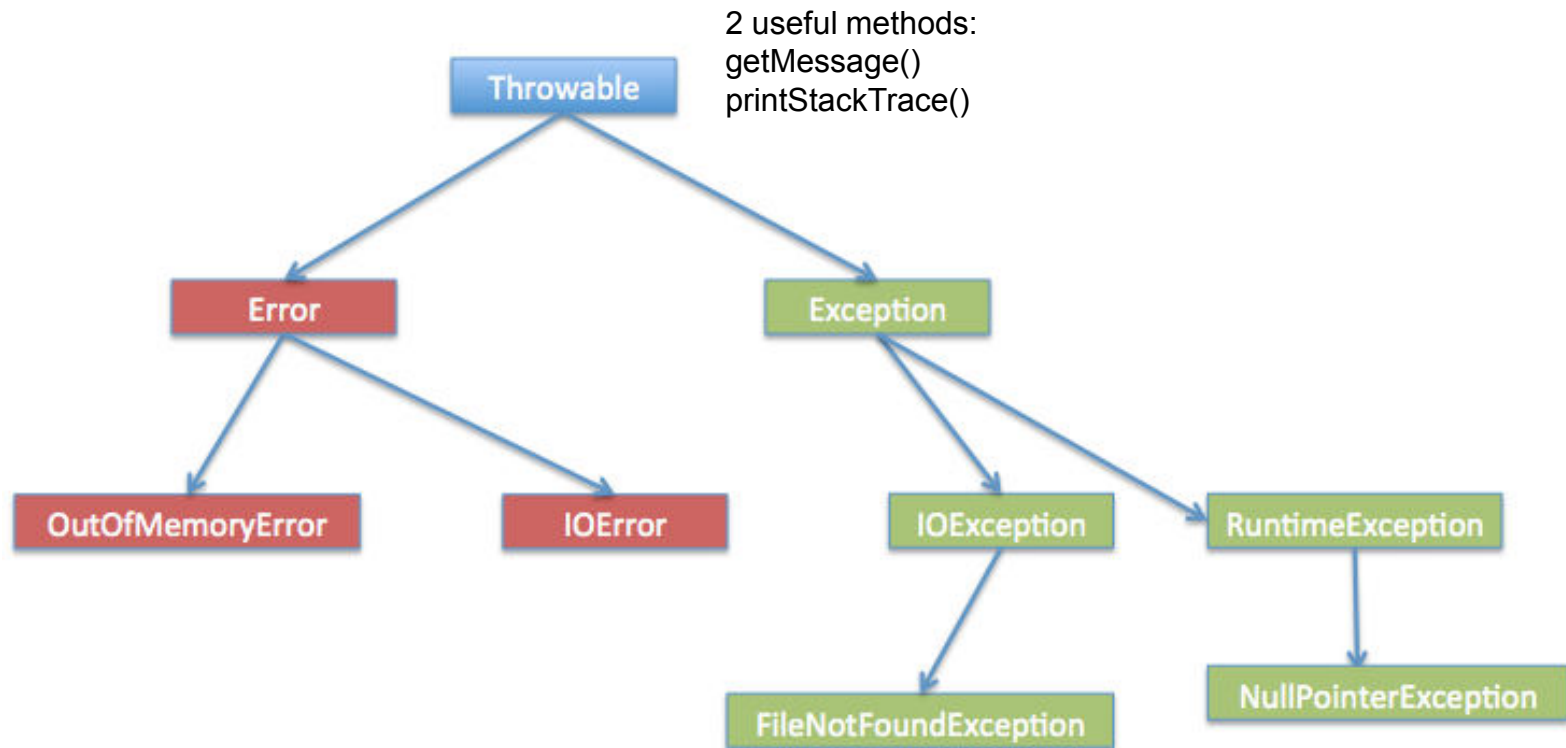
```
<terminated> TestFinally [Java Application] C:\Program Fil  
Catch from test: throwing an exception  
Finally from test method.
```





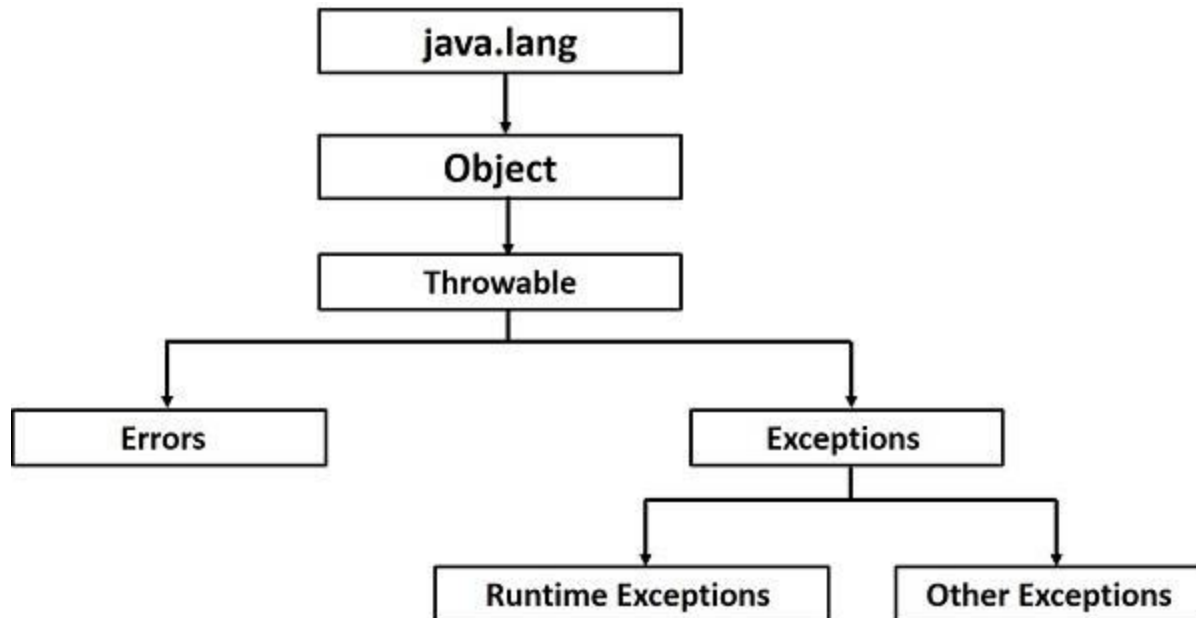
# EXCEPTION HIERARCHY

- All exception classes are subtypes of java.lang.Exception class.



# EXCEPTION HIERARCHY

- All exception classes are subtypes of `java.lang.Exception` class.



# TYPES OF EXCEPTION

- 2 types
  - Checked
  - Unchecked



# CHECKED EXCEPTION

- Checked exceptions are **checked at compile-time**.
- It means if a method is throwing a checked exception then we cannot ignore at compile time
- it should handle the exception using **try-catch block** or it should declare the exception using **throws keyword**,
- otherwise the program will give a compilation error.
- It is named as ***checked exception*** because these exceptions are ***checked*** at Compile time.



# UNCHECKED EXCEPTION

- Unchecked exceptions are **not checked at compile time**.
- If a program is throwing an unchecked exception, the program won't give a compilation error if you didn't handle/declare that exception.
- Most of the time these exceptions occur due to
  - bad data provided by user during the user-program interaction.
  - Logical error.
- All Unchecked exceptions are direct sub classes of **RuntimeException** class.



# ERROR

- These are not exceptions at all, but problems that arise beyond the control of the user or the programmer.
- Errors are typically ignored in your code because you can rarely do anything about an error.
- For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.



# CATCHING SUBCLASS EXCEPTION

- You can not catch the same exception twice at same level.
- If a catch block is written to handle a super class exception, it will also catch subclass exception.

```
try{  
    }catch(Exception e){ // handle IOException as well  
  
    }catch(IOException e) { // Compile error: Unreachable catch block for IOException.  
        It is already handled by the catch block for Exception  
    }
```

- To handle the subclass separately, it must appear a catch block before the parent catch block

```
try{  
    }catch(IOException e){ //OK,  
  
    }catch(Exception e) { // All exception except IOException will be cathed here  
    }
```



# USER DEFINED EXCEPTION





# USER DEFINED EXCEPTION

- Sometimes you may need to create your own exception.
- How?
  - just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**).
  - Your subclasses don't need to actually implement anything—it is their existence in the **type system** that allows you to use them as exceptions.



# USER DEFINED EXCEPTION

- Exception class doesn't have any method, only constructors.
  - The most commonly used constructors are.
    - `Exception()`
    - `Exception(String msg)`
- Once you create the exception, you can **throw** and **catch** the exception as any other Exception.



# EXAMPLE - USER DEFINED EXCEPTION

```
class SuperHeroException extends Exception {  
    public SuperHeroException() {  
        super();  
    }  
  
    public SuperHeroException(String message) {  
        super(message);  
    }  
  
    public SuperHeroException(int energyLevel) {  
        super("Energy level dropped below:" + energyLevel);  
    }  
}
```



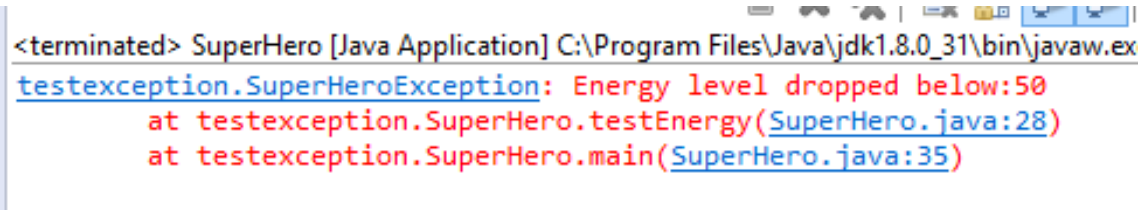
# EXAMPLE - USER DEFINED EXCEPTION

```
public class SuperHero {
    int energyLevel;
    public SuperHero(int a){
        energyLevel = a;
    }

    public void testEnergy() throws SuperHeroException {
        if(energyLevel < 50)
            throw new SuperHeroException(50);
    }

    public static void main(String[] args){
        SuperHero hero = new SuperHero(40);
        try{
            hero.testEnergy();
        }catch(SuperHeroException e){
            e.printStackTrace();
        }
    }
}
```

## ○ Output



```
<terminated> SuperHero [Java Application] C:\Program Files\Java\jdk1.8.0_31\bin\javaw.exe
testexception.SuperHeroException: Energy level dropped below:50
    at testexception.SuperHero.testEnergy(SuperHero.java:28)
    at testexception.SuperHero.main(SuperHero.java:35)
```

## EXAMPLE - USER DEFINED EXCEPTION

- See the word doc for another example related to BankAccount



# REFERENCE

- Java: Complete Reference - Chapter 10
- Java: How to Program – Chapter 11
- Online Reference:
  - [http://www.tutorialspoint.com/java/java\\_exceptions.htm](http://www.tutorialspoint.com/java/java_exceptions.htm)

