Exception Handling

Reference: Liang, Y., 2015. Intro To Java Programming. 10th ed. Pearson Education Limited.

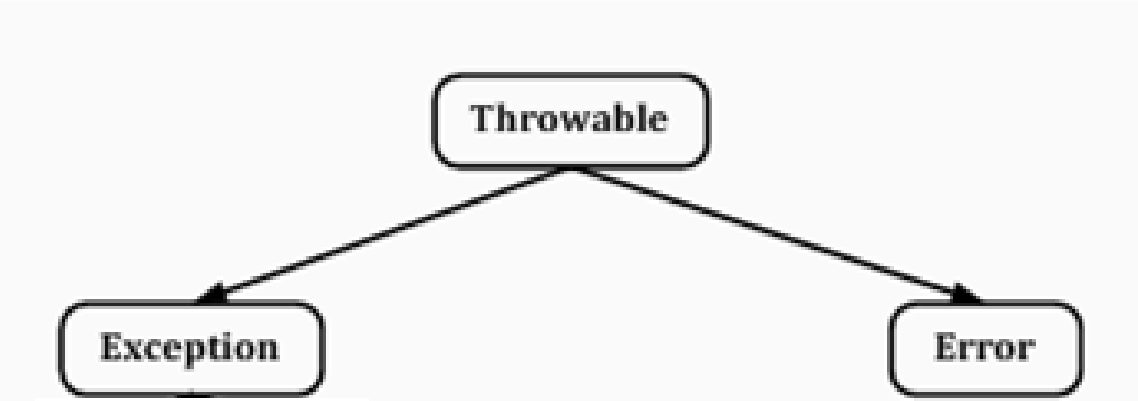
Exception-Handling

What is an **exception**?

- An exception is an abnormal condition that arises in a code sequence at run time.
- An exception is a run-time error.
- Exception handling avoids these problems and, in the process, brings run-time error management into the object oriented world.
- Exception is an object that describes an exceptional 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. This exception is then caught and processed.

Throwable

- **Throwable** is at the top of the Exception class hierarchy.
- All exception types are subclasses of the built-in class Throwable.
- Throwable contains two subclasses
  1. Exception
  2. Error



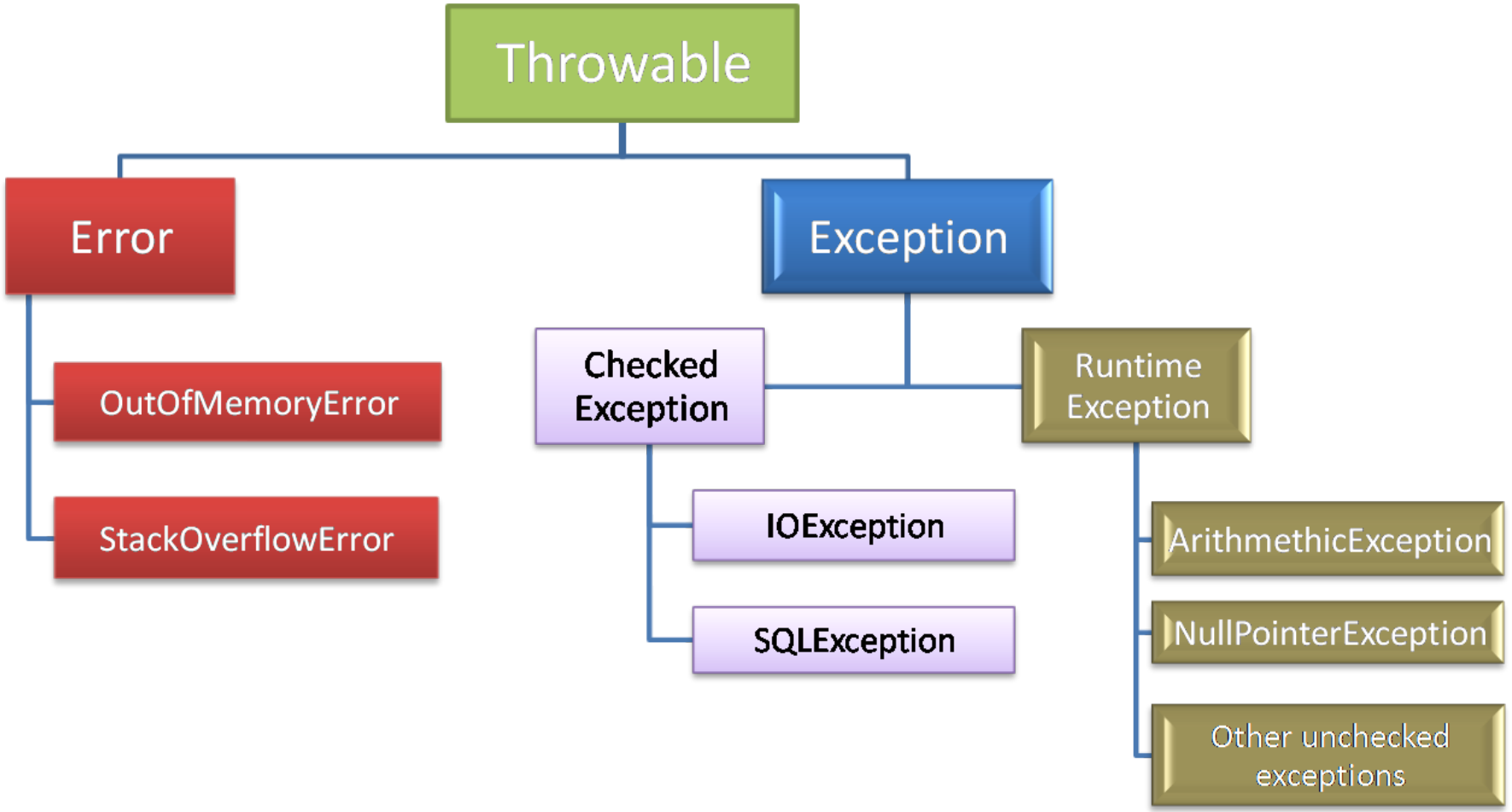
Error

- One branch of Throwable is topped by **Error** class, which defines exceptions that are not expected to be caught under normal circumstances by a program or application.
- Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself.
- Example:
  - Stack overflow
  - Out of Memory
- These are typically created in response to catastrophic failures that cannot usually be handled by a program.

Exception

- The second branch of Throwable is headed by **Exception**. This class is used for exceptional conditions that user programs should catch.
- Exception can further be classified into following 2 broad categories.
  - **Unchecked or RuntimeException**
    - Exceptions of this type are automatically defined for the programs that is being written like division by zero and invalid array indexing.
    - These are called unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions.
  - **Checked Exception**
    - Checked Exception must be included in a method's throws list if that method can generate one of these exceptions and does not handle it itself.

Exception Types



Uncaught Exceptions

```
class Exc1 {
        public static void main(String args[]) {
                int d = 0;
                int a = 42 / d;
        }
}
```

This small program includes an expression that intentionally causes a divide-by-zero error

```
class Exc1 {
  static void subroutine() {
    int d = 0;
    int a = 10 / d;
  }
  public static void main(String args[]) {
    Exc1.subroutine();
  }
}
```

Output:

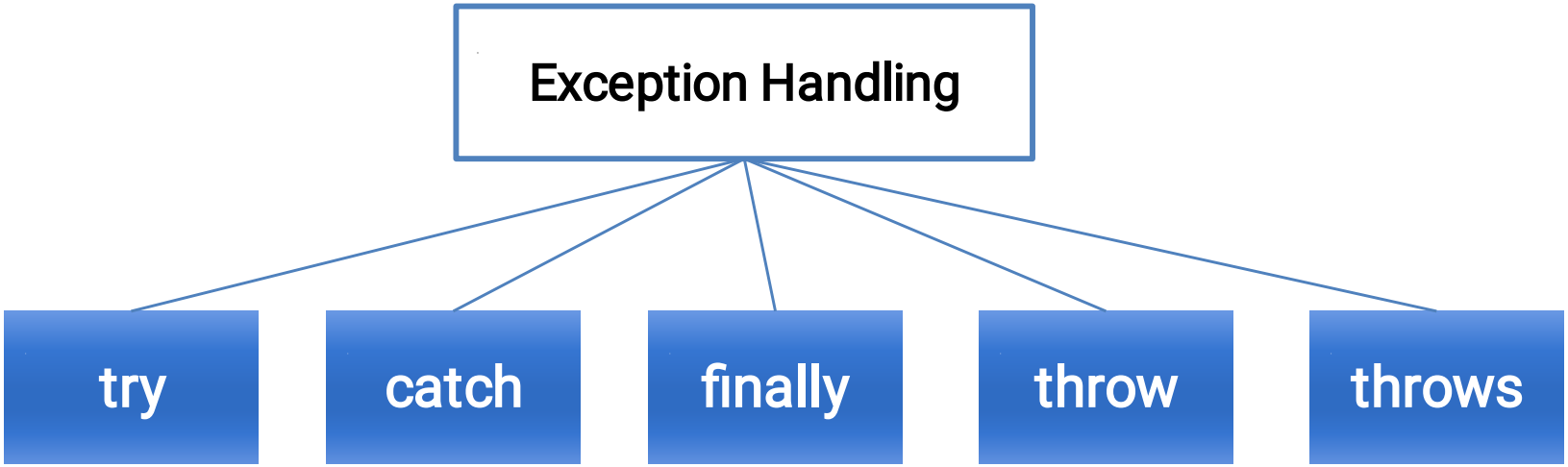
```
java.lang.ArithmeticException: / by zero
     at Exc1.subroutine(Exc1.java:4)
     at Exc1.main(Exc1.java:7)
```

How an exception is propagated?

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. Once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.
- In this example, exception handlers is not defined, so the exception is caught by the default handler provided by the Java run-time system.
- The default handler displays a string describing the exception, prints a **stack trace** from the point at which the exception occurred, and terminates the program.

Exception handling in Java

- Although the default exception handler is useful for debugging, but exceptions can also be handled by program for two reason;
  - it allows you to fix the error.
  - it prevents the program from automatically terminating.
- Java exception handling is managed via **five keywords**:



try

- The **try** statement allows you to define a block of code to be tested for errors while it is being executed.
- 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 th

```
try {
   // block of code to monitor for errors
}
```

- The statements that are protected by try must be surrounded by curly braces. (That is, they must be within a block.) You cannot use try on a single statement.

catch

- The **catch** statement allows you to define a block of code to be executed, if an error occurs in the **try** block.
- Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch.

```
catch (ExceptionType1 exOb) {
   // exception handler for ExceptionType1
}
```

- ExceptionType is the type of exception that has occurred.
- The scope of the catch clause is restricted to those statements specified by the immediately preceding try statement. A catch statement cannot catch an exception thrown by another try statement.
- The goal of most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.

Using try-catch

```
class Exc2 {
  public static void main(String args[]) {
    int d, a;

    try { // monitor a block of code.
      d = 0;
      a = 42 / d;
      System.out.println("After Exception ");
    } catch (ArithmeticException e) { // catch divide-by-zero error
      System.out.println("Division by zero.");
    }
    System.out.println("After catch statement.");
  }
}
```

Output

```
Division by zero.
After catch statement.
```

- In the example, the call to println( ) inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block.
- Put differently, catch is “called,” so execution never “returns” to the try block from a catch. Thus, the line “After Exception ” is not displayed.
- Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.

Displaying Exception Description

- Throwable overrides the toString( ) method (defined by Object) so that it returns a string containing a description of the exception. You can display this description in a println( ) statement by simply passing the exception as an argument.

```
catch (ArithmeticException e) {
  System.out.println("Exception: " + e);
  a = 0; // set a to zero and continue
}
```

Output

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

IQRA UNIVERSITY IU

Multiple catch Clauses

- In some cases, more than one exception could be raised by a single piece of code.
- Specify two or more catch clauses, each catching a different type of exception.
- When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.

Multiple catch Clauses

```
import java.io.*;
public class Read{
public static void main(String args[])
{
try {
int a=0;
System.out.println("a="+a);
int b= 42/a;
int c[]= {1};
c[2]=50;
}
catch(ArithmeticException e)
{
System.out.println("Divide by zero: "+e);
}
catch(ArrayIndexOutOfBoundsException e )
{System.out.println("Array out of Bound: "+ e);
}
System.out.println("After try/catch blocks");
}
```

```
import java.io.*;
public class Read{
public static void main(String args[])
{
try {
int a=1;
System.out.println("a="+a);
int b= 42/a;
int c[]= {1};
c[2]=50;
}
catch(ArithmeticException e)
{
System.out.println("Divide by zero: "+e);
}
catch(ArrayIndexOutOfBoundsException e )
{System.out.println("Array out of Bound: "+ e);
}
System.out.println("After try/catch blocks");
}
```

Output:

a=0
Divide by zero: java.lang.ArithmeticException: / by zero
After try/catch blocks

a=1
Array out of Bound: java.lang.ArrayIndexOutOfBoundsException: 2
After try/catch blocks

Multiple catch Clauses

- When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their superclasses.

```
import java.io.*;
public class MultiCatch{
public static void main(String args[])
{
try {
int a =0;
int b= 42/a;
}
catch(ArithmeticException e)
{
System.out.println("Arithmetic Exception: "+e);
}
catch(Exception e )
{System.out.println("Generic Exception: "+ e);
}
}
```

```
import java.io.*;
public class MultiCatch{
public static void main(String args[])
{
try {
int a =0;
int b= 42/a;
}
catch(Exception e )
{System.out.println("Generic Exception: "+ e);
}
catch(ArithmeticException e)//Error
{
System.out.println("Arithmetic Exception: "+e);
}

}
```

Output:
Arithmetic Exception: java.lang.ArithmeticException: / by zero

Nested try Statements

- The try statement can be nested.
- That is, a try statement can be inside the block of another try. If an inner try statement does not have a catch handler for a particular exception, the next try statement's catch handlers are inspected for a match.
- This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.
- If no catch statement matches, then the Java run-time system will handle the exception.
- Nesting of try statements can also occur when method calls are involved. For example, you can enclose a call to a method within a try block. Inside that method is another try statement. In this case, the try within the method is still nested.

Nested try Statements

```
try {
        try {
                int a = 10;
                a = 10 / 0;
        } catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("catch 1 “+e);
        }
}
catch (ArithmeticException e) {
        System.out.println(“ catch 2“+e);
}
```

Practice!

- Write a simple java program that adds two integers input by user. The program should terminate if any operand is nonnumeric. Before program exit, a message should be displayed that “Input is not valid! ”.

  Hint: You can use
  - InputMismatchException

```
import java.util.InputMismatchException;
import java.io.*;
import java.util.Scanner;
public class add{
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int num1=0,num2=0,r=0;
      try {
           System.out.println("Enter Integer Value1: ");
          num1 = sc.nextInt();
         try {  System.out.println("Enter Integer Value2: ");
          num2 = sc.nextInt();
          r=num1+num2;
          System.out.println(num1 + "+" + num2 + "=" + r );
        }
        catch (InputMismatchException e) {
        System.out.println("input is not valid"+ e);
        }
         }
         catch (Exception e) {
         System.out.println("input is not valid"+ e);
         }
sc.close();
   }
```

Output:

Enter Integer Value1:
x
input is not valid
java.util.InputMismatchException

Output:

Enter Integer Value1:
22
Enter Integer Value2:
x
input is not valid
java.util.InputMismatchException

Output:

Enter Integer Value1:
22
Enter Integer Value2:
22
22+22=44

finally

- Any code that absolutely must be executed after a try block completes is put in a ***finally*** block.

```
finally {
    // block of code to be executed after try block ends
}
```

- ***finally*** creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block.
- The ***finally*** block will execute whether or not an exception is thrown.
- If an exception is thrown, the finally block will execute even if no catch statement matches the exception.
- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.
- A try and its catch along with finally statement form a unit. The finally clause is optional. However, each try

```
try {
    // block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
    // exception handler for ExceptionType1
}

finally {
    // block of code to be executed after try block ends
}
```

finally

```
import java.io.*;
public class Read{
public static void main(String args[]) {
try {
procA();
}
catch(Exception e)
{
System.out.println("Exception caught: "+e);
}
procB();
procC();
}
static void procA() {
try {
 System.out.println("inside procA");
 int I =10 / 0;
 }
finally {
 System.out.println("procA's finally");
 }
 }
static void procB() {
try {
 System.out.println("inside procB");
}
finally {
 System.out.println("procB's finally");
}
 }
```

```
static void procC() {
try {
 System.out.println("inside procC");
}
finally {
 System.out.println("procC's finally");
}
 }
```

Output:

```
inside procA
procA's finally
Exception caught:java.lang.ArithmeticException:/by zero
inside procB
procB's finally
inside procC
procC's finally
```

Custom Exception

You can define a custom exception class by extending the java.lang.Exception class. Java provides quite a few exception classes. Use them whenever possible instead of defining your own exception classes. However, if you run into a problem that cannot be adequately described by the predefined exception classes, you can create your own exception class, derived from Exception or from a subclass of Exception, such as IOException

```
1  package sqcu;
2
3  class UserException extends Exception {
4          public UserException() {
5          }
6          UserException(String s){
7        super(s);
8          }
9  }
```

```
1  package sqcu;
2  import java.io.*;
3  import java.util.*;
4          public class ExceptionTest {
5             public void method() throws UserException{
6              System.out.println("Throwing UserException from method()");
7            throw new UserException("Exception thrown from method()");
8              }
9             public static void main(String[] args) {
10           ExceptionTest test = new ExceptionTest();
11           try{
12                    test.method();
13           }catch(UserException e){
14              e.printStackTrace();
15           }
16             }
```

```
1  package sqcu;
2  import java.io.*;
3  import java.util.*;
4          public class ExceptionTest {
5             public void method() throws UserException{
6              System.out.println("Throwing UserException from method()");
7            throw new UserException("Exception thrown from method()");
8              }
9             public static void main(String[] args) {
10           ExceptionTest test = new ExceptionTest();
11           try{
12                    test.method();
13           }catch(UserException e){
14              e.printStackTrace(System.out);
15           }
16             }
```

```
Throwing UserException from method()
sqcu.UserException: Exception thrown from method()
        at sqcu.ExceptionTest.method(ExceptionTest.java:7)
        at sqcu.ExceptionTest.main(ExceptionTest.java:12)
```

```
Throwing UserException from method()
sqcu.UserException: Exception thrown from method()
        at sqcu.ExceptionTest.method(ExceptionTest.java:7)
        at sqcu.ExceptionTest.main(ExceptionTest.java:12)
```

Practice!

Write a class **InvalidCNICException** that extends **Exception** class.

The class should have following details implemented

- Two constructors
  1. With no parameter
  2. With String parameter

Take user input for CNIC. The CNIC is considered valid if

1. The length of CNIC is 15(including -)
2. if the hyphen(-) is present at position 6 and 14.

Generate (throw) **InvalidCNICException** when an invalid CNIC has been entered by the user. Handle this exception by giving user a responsive message saying "The CNIC you entered is not valid"

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.Scanner;
public class ValidateCNIC {
  public void method() {
 String CNIC;// = "42101-9468999-2";
        Scanner sc= new Scanner(System.in);
        System.out.println("Enter CNIC in the following pattern XXX-XXXXXXX-X");
        CNIC=sc.next();
  Pattern pattern = Pattern.compile("\\d{5}-\\d{7}-\\d{1}");
    Matcher matcher = pattern.matcher(CNIC);
    try
    {if (matcher.matches())
    System.out.println("CNIC Valid");
    else throw new InvalidCNICException("The CNIC you Entered is not valid");
    }
    catch(InvalidCNICException e) {
    System.out.println("The CNIC you Entered is not valid");
    }
    }
  public static void main(String[] args) {
 ValidateCNIC test = new ValidateCNIC();
 test.method();
  }
 }
```

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

```
Enter CNIC in the following pattern XXX-XXXXXXX-X
12345-6789123-2
CNIC Valid
```

```
Enter CNIC in the following pattern XXX-XXXXXXX-X
561456655656
The CNIC you Entered is not valid
```

Design a program to implement calculator's basic functionality with an exception handler that deals with in correct operands. Your program should display a message that informs the user of the wrong operator type before exiting.

```
import java.util.Scanner;
public class Main {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
     System.out.println("Operand1:");
      int a = sc.nextInt();
      System.out.println("Operand2:");
      int b = sc.nextInt();
try {
      System.out.println("Operator(+ - / *):");
      char operator = sc.next().charAt(0);
      if(operator=='+')
      System.out.println(a + b);
      else if(operator=='-')
      System.out.println(a - b);
      else if(operator=='*')
      System.out.println(a * b);
      else if(operator=='/')
      {if(b==0)
       System.out.println("Can not divide with zero");
      else
      System.out.println(a / b);}
  else throw new InvalidoperatorException("The operator you Entered is not valid");
      }
    catch (InvalidoperatorException e) {
    System.out.println("Invalid operator");
    }
  }
}
```

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

Review Questions

1. What is Exception?
2. Explain the advantages of using exception handling
3. Difference between Error and exception class
4. Difference between Checked and Uncheck exception class
5. Explain how an exception is propagated
6. Define how the exception can be handled?
7. How can you create custom exception?