CSIT121
Object Oriented Design and Programming

Lesson 5

Exception Handling and Assertion

Exception

- Problem that arises during the execution of a program.
- Exception occur for many different reasons, examples::
  - A user has entered invalid data.
  - A file that needs to be opened cannot be found.
  - A network connection has been lost
  - JVM has run out of memory.

Categories of exceptions

- Checked exceptions:
  - Typically, a user error or a problem that cannot be foreseen by the programmer.
  - For example, if a file is to be opened cannot be found, an exception occurs.
  - These exceptions must be handled at the time of compilation.

Categories of exceptions

- Runtime exceptions (Unchecked Exception):
  - An exception that occurs that could have been avoided by the programmer by doing more checks.
  - Runtime exceptions are ignored at the time of compilation.

Error

- Error:
  - Not exceptions,
  - Problems that arise beyond the control of the user or the programmer.
  - Typically ignored in code because we can rarely do anything about an error.
  - For example, if a stack overflow occurs and out of memory, an error will arise.
  - Ignored at the time of compilation.

java.lang.Object
    java.lang.Throwable
        java.lang.Exception

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AclNotFoundException, ActivationException, AlreadyBoundException, ApplicationException, AWTException, BackingStoreException, BadAttributeValueExpException, BadBinaryOpValueExpException, BadLocationException, BadStringOperationException, BrokenBarrierException, CertificateException, CloneNotSupportedException, DataFormatException, DatatypeConfigurationException, DestroyFailedException, ExecutionException, ExpandVetoException, FontFormatException, GeneralSecurityException, GSSException, IllegalClassFormatException, InterruptedException, IntrospectionException, InvalidApplicationException, InvalidMidiDataException, InvalidPreferencesFormatException, InvalidTargetObjectTypeException, IOException, JAXBException, JMException, KeySelectorException, LastOwnerException, LineUnavailableException, MarshalException, MidiUnavailableException, MimeTypeParseException, MimeTypeParseException, NamingException, NoninvertibleTransformException, NotBoundException, NotOwnerException, ParseException, ParserConfigurationException, PrinterException, PrintException, PrivilegedActionException, PropertyVetoException, ReflectiveOperationException, RefreshFailedException, RemarshalException, RuntimeException, SAXException, ScriptException, ServerNotActiveException, SOAPException, SQLException, TimeoutException, TooManyListenersException, TransformerException, TransformException, UnmodifiableClassException, UnsupportedAudioFileException, UnsupportedCallbackException, UnsupportedFlavorException, UnsupportedLookAndFeelException, URIReferenceException, URISyntaxException, UserException, XAException, XMLParseException, XMLSignatureException, XMLStreamException, XPathException

```
public class Exception
extends Throwable
```

The class `Exception` and its subclasses are a form of `Throwable` that indicates conditions that a reasonable application might want to catch.

The class `Exception` and any subclasses that are not also subclasses of `RuntimeException` are *checked exceptions*. Checked exceptions need to be declared in a method or constructor's `throws` clause if they can be thrown by the execution of the method or constructor and propagate outside the method or constructor boundary.

RuntimeException

java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AnnotationTypeMismatchException, ArithmeticException, ArrayStoreException, BufferOverflowException, BufferUnderflowException, CannotRedoException, CannotUndoException, ClassCastException, CMMException, ConcurrentModificationException, DataBindingException, DOMException, EmptyStackException, EnumConstantNotPresentException, EventException, FileSystemAlreadyExistsException, FileSystemNotFoundException, IllegalArgumentException, IllegalMonitorStateException, IllegalPathStateException, IllegalStateException, IllformedLocaleException, ImagingOpException, IncompleteAnnotationException, IndexOutOfBoundsException, JMRuntimeException, LSException, MalformedParameterizedTypeException, MirroredTypesException, MissingResourceException, NegativeArraySizeException, NoSuchElementException, NoSuchMechanismException, NullPointerException, ProfileDataException, ProviderException, ProviderNotFoundException, RasterFormatException, RejectedExecutionException, SecurityException, SystemException, TypeConstraintException, TypeNotPresentException, UndeclaredThrowableException, UnknownEntityException, UnmodifiableSetException, UnsupportedOperationException, WebServiceException, WrongMethodTypeException

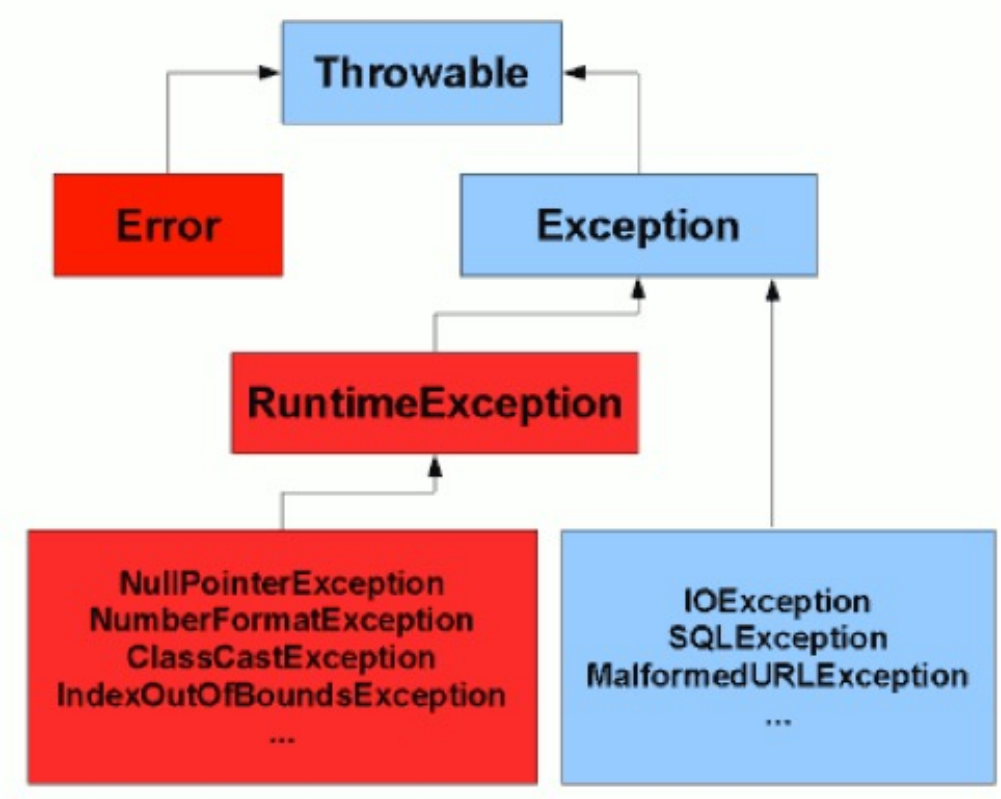
---

```
public class RuntimeException
extends Exception
```

`RuntimeException` is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.

`RuntimeException` and its subclasses are *unchecked exceptions*. Unchecked exceptions do *not* need to be declared in a method or constructor's `throws` clause if they can be thrown by the execution of the method or constructor and propagate outside the method or constructor boundary.

Exception Hierarchy



Error, RuntimeException and their subclasses are **unchecked**
Exception and subclasess are **checked.**

Exception Hierarchy

- Checked Exception MUST be handled in the codes
- RunTimeException (Unchecked Exception) and Error need not be handled in the codes.
- However, we can choose to handle RunTimeException in the codes too
- We don’t usually handle Error in the codes
  - Error occur in case of severe failures, which almost are not possible to be handled in the codes.
  - Errors are generated to indicate errors generated by the runtime environment. Example : JVM is out of Memory. Normally programs cannot recover from errors

Handling an Exception

- A program can be designed to process an exception in one of the three ways:
  - do not handle the exception at all
  - handle the exception where it occurs, or
  - handle the exception at another point in the program (by propagating the exception)

Catching Exception

- A try/catch block is placed around the code that might generate an exception.
- Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
   //Protected code
}catch(ExceptionName e1)
{
   //Catch block
}
```

FileNotFoundException

- Is a type of IOException (Checked Exception)
- Any method that throw this exception, MUST handle this exception

java.io

Class FileNotFoundException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.io.IOException
        java.io.FileNotFoundException

```
public class FileNotFoundException
extends IOException
```

Signals that an attempt to open the file denoted by a specified pathname has failed.

This exception will be thrown by the `FileInputStream`, `FileOutputStream`, and `RandomAccessFile` constructors when a file with the specified pathname does not exist. It will also be thrown by these constructors if the file does exist but for some reason is inaccessible, for example when an attempt is made to open a read-only file for writing.

FileNotFoundException

- We use a Scanner to read in a file

Scanner

```
public Scanner(File source)
        throws FileNotFoundException
```

Constructs a new `Scanner` that produces values scanned from the specified file. Bytes from the file are converted into characters using the underlying platform's default charset.

Parameters:

`source` - A file to be scanned

Throws:

`FileNotFoundException` - if source is not found

FileNotFoundException

```
try {
    Scanner myScanner = new Scanner(new File("studentsMarks.dat"));
    String data;

    while(myScanner.hasNextLine()){
        data = myScanner.nextLine();
        System.out.println(data);
    }

}catch(FileNotFoundException ex){
    System.out.println("File not found");
}
```

ArrayIndexOutOfBoundException

- Is a type of RuntimeException (Unchecked Exception)
- Need not handle this exception, but we can choose to handle

Class ArrayIndexOutOfBoundsException

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IndexOutOfBoundsException
                    java.lang.ArrayIndexOutOfBoundsException
```

```
public class ArrayIndexOutOfBoundsException
extends IndexOutOfBoundsException
```

Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

ArrayIndexOutOfBoundException

```
int[] num = {1,2,3,4};

for(int i=0;i<=4;i++){
    System.out.println(num[i]);
}
```

```
java.lang.ArrayIndexOutOfBoundsException: 4
        at StudentTest.test(StudentTest.java:49)
        at StudentTest.main(StudentTest.java:58)
```

InputMismatchException

Class InputMismatchException

java.lang.Object
- java.lang.Throwable
  - java.lang.Exception
    - java.lang.RuntimeException
      - java.util.NoSuchElementException
        - java.util.InputMismatchException

```
public class InputMismatchException
extends NoSuchElementException
```

Thrown by a `Scanner` to indicate that the token retrieved does not match the pattern for the expected type, or that the token is out of range for the expected type.

InputMismatchException

```
import java.util.Scanner;
public class InputException{
    public static void main (String[] args) {
        Scanner myScanner= new Scanner (System.in);
        System.out.println("Enter a number ");
        int num = myScanner.nextInt();
    }
}
```

User may not enter an integer value

General Output

```
abc
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:840)
    at java.util.Scanner.next(Scanner.java:1461)
    at java.util.Scanner.nextInt(Scanner.java:2091)
    at java.util.Scanner.nextInt(Scanner.java:2050)
    at InputException.main(InputException.java:6)
```

Class View | Package View | Find Results | Task View | General Output | Build Output

NumberFormatException

Class NumberFormatException

```
java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.IllegalArgumentException
                    java.lang.NumberFormatException
```

```
public class NumberFormatException
extends IllegalArgumentException
```

Thrown to indicate that the application has attempted to convert a string to one of the numeric types, but that the string does not have the appropriate format.

NumberFormatException

```
import java.util.Scanner;
public class NumberFormatException{
    public static void main (String[] args) {
        Scanner myScanner= new Scanner (System.in);
        System.out.println("Enter a number ");
        String input = myScanner.nextLine();
        Int num = Integer.parseInt(input);
    }
}
```

Input cannot be converted to an int value

General Output

```
Enter a number
abc
Exception in thread "main" java.lang.NumberFormatException: For input string: "abc"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
    at java.lang.Integer.parseInt(Integer.java:447)
    at java.lang.Integer.parseInt(Integer.java:497)
    at NumberFormatException.main(NumberFormatException.java:11)
```

Class View | Package View | Find Results | Task View | General Output | Build Output

NullPointerException

Class NullPointerException

java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
                java.lang.NullPointerException

```
public class NullPointerException
extends RuntimeException
```

Thrown when an application attempts to use `null` in a case where an object is required. These include:

- Calling the instance method of a `null` object.
- Accessing or modifying the field of a `null` object.
- Taking the length of `null` as if it were an array.
- Accessing or modifying the slots of `null` as if it were an array.
- Throwing `null` as if it were a `Throwable` value.

NullPointerException

```java
String[] items = new String[3];
items[0] = "apple";
items[2] = "orange";

for(String item: items) {
    System.out.println(item);
}
System.out.println();

for(String item: items) {
    System.out.println(item.length());
}
```

```
apple
null
orange

5
Exception in thread "main" java.lang.NullPointerException
        at NullExample.main(NullExample.java:17)
```

Finally

- Used to create a block of code that follows a try block.
- A finally block of code always executes, whether an exception has occurred.
- Using used to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

```
try
{
   //Protected code
}catch(ExceptionType1 e1)
{
   //Catch block
}catch(ExceptionType2 e2)
{
   //Catch block
}catch(ExceptionType3 e3)
{
   //Catch block
}finally
{
   //The finally block always executes.
}
```

Finally

```
int a[] = new int[2];
try{
   System.out.println("Access element three :" + a[3]);
}catch(ArrayIndexOutOfBoundsException e){
   System.out.println("Exception thrown  :" + e);
}
finally{
   a[0] = 6;
   System.out.println("First element value: " +a[0]);
   System.out.println("The finally statement is executed");
}
```

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

Finally and return

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

    System.out.println(testIfFileExist());
}

public static boolean testIfFileExist(){

    try{

        Scanner file = new Scanner(new File ("data.dat"));
        return true;
    }catch(FileNotFoundException ex){
        System.out.println("File Not Found");
        return false;
    }finally{
        System.out.println("Finally");
    }
}
```

Output if file found

```
Finally
true
```

Output if file not found

```
File Not Found
Finally
false
```

Method throws Exception

- If a method does not handle a checked exception, the method must declare it using the throws keyword.
- The throws keyword appears at the end of a method's signature.

```
public static void main(String []args) throws FileNotFoundException
{
    //This might throw an exception if file is not found
    //It will be throw to the main method and display on the console
    Scanner fileScanner = new Scanner(new File("studentMarks.dat"));
}
```

Method throws Exception

- A method can throws more than one exceptions.
- In this case the exceptions are declared in a list separated by commas.

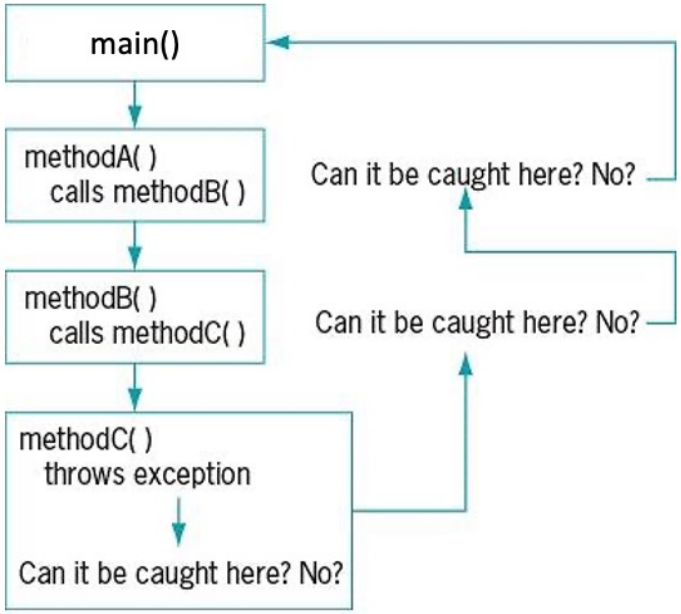
```
public void withdraw(double amount) throws RemoteException,
                                           InsufficientFundsException
{
    // Method implementation
}
```

Exception Propagation

- If an exception is not caught and handled where it occurs, control is immediately returned to the method that invoked the method that produced the exception.
- The propagation continues until the exception is caught and handled or until it is passed out of the main method, which terminates the program and produces the exception message

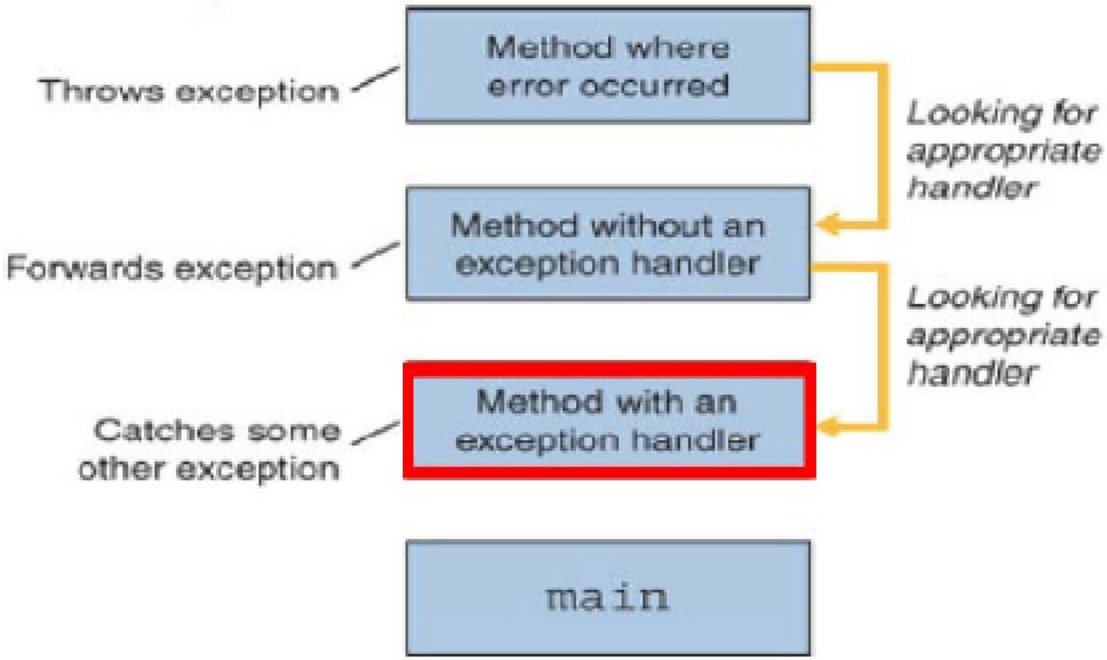
Exception Propagation

- It searches the ordered list of methods that had been called to get to the method where the error occurred.
  - The list of methods is known as the “call stack
- The runtime system searches the call stack for a method that contains an exception handler for the given exception.



Exception Propagation

- The exception handler chosen is said to catch the exception.



Exception Propagation Example 1

```java
import java.io.*;
import java.util.*;

public class ExceptionEg1 {

    public static void main(String []args) throws FileNotFoundException
    {
        //This might throw an exception if file is not found
        //It will be throw to the main method and display on the console
        Scanner fileScanner = new Scanner(new File("studentMarks.dat"));
    }
}
```

```
Exception in thread "main" java.io.FileNotFoundException: studentMarks.dat (No such file or directory)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:120)
        at java.util.Scanner.<init>(Scanner.java:636)
        at ExceptionEg1.main(ExceptionEg1.java:10)
```

Exception Propagation Example 2

```java
public class ExceptionEg2 {

    public static void main(String[] args) {

        try{
            //This might throw an exception if file is not found
            //This exception will be caught by the catch statement below
            Scanner fileScanner = new Scanner(new File("studentMarks.dat"));

        }catch(FileNotFoundException ex)
        {
            //catch the file not found exception thrown by scanner
            System.out.println("File not found");
        }
    }

}
```

```
File not found
```

Exception Propagation Example 3

```java
public class ExceptionEg3 {

    public static void main(String[] args) throws FileNotFoundException{

        //This method might throw FileNotFoundException
        //Since the exception is not caught, it will be throw
        //to the main and then to the console
        readFile();
    }

    public static void readFile() throws FileNotFoundException
    {
        //This might throw an exception if file is not found
        //This exception will thrown to the calling method
        Scanner fileScanner = new Scanner(new File("studentMarks.dat"));
    }
}
```

```
Exception in thread "main" java.io.FileNotFoundException: studentMarks.dat (No such file or directory)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:120)
        at java.util.Scanner.<init>(Scanner.java:636)
        at ExceptionEg3.readFile(ExceptionEg3.java:18)
        at ExceptionEg3.main(ExceptionEg3.java:11)
```

Exception Propagation Example 4

```java
public class ExceptionEg4 {

    public static void main(String[] args){

        //This method will not throw any Exception as it is already handled in the method
        readFile();
    }

    public static void readFile() //don't need to throw any exception as it has been caught
    {
        try{
            //This might throw an exception if file is not found
            //This exception will be caught by the catch statement below
            Scanner fileScanner = new Scanner(new File("studentMarks.dat"));

        }catch(FileNotFoundException ex)
        {
            //catch the file not found exception thrown by scanner
            System.out.println("File not found");
        }
    }
}
```

```
File not found
```

Exception Propagation Example 5

```java
public class ExceptionEg5 {

    public static void main(String[] args) {

        try {

            // This exception will be caught by the catch statement below
            readFile();

        } catch (FileNotFoundException ex) // catch the file not found exception
        {
            System.out.println("File not found");
        }
    }

    public static void readFile() throws FileNotFoundException {
        // This might throw an exception if file is not found
        // The exception is not caught here and it is throw to the calling method
        Scanner fileScanner = new Scanner(new File("studentMarks.dat"));
    }
}
```

```
File not found
```

Exception Propagation Example 6

```java
public class ExceptionEg6 {

    public static void main(String[] args) {

        readFile1();
    }

    public static void readFile1(){
        try {

            // This exception will be caught by the catch statement below
            readFile2();

        } catch (FileNotFoundException ex) // catch the file not found exception
        {
            System.out.println("File not found");
        }
    }

    public static void readFile2() throws FileNotFoundException {
        // This might throw an exception if file is not found
        // The exception is not caught here and it is throw to the calling method
        Scanner fileScanner = new Scanner(new File("studentMarks.dat"));
    }
}
```

```
File not found
```

Exception Propagation Example 7

```java
public class ExceptionEg7 {

    public static void main(String[] args) {

        readFile1();
    }


    public static void readFile1(){
        try {

            // This exception will be caught by the catch statement below
            readFile2();

        } catch (FileNotFoundException ex) // catch the file not found exception
        {
            System.out.println("File not found");
        }
        finally //will always come here in regardless if the exception occurs a not
        {
            System.out.println("Finally");
        }
    }

    public static void readFile2() throws FileNotFoundException {
        // This might throw an exception if file is not found
        // The exception is not caught here and it is throw to the calling method
        Scanner fileScanner = new Scanner(new File("test1.txt"));
    }
}
```

```
File not found
Finally
```

Exception Propagation Example 8

```java
public class ExceptionEg8 {

    public static void main(String[] args) {

        readFile1();
    }

    public static void readFile1(){
        try {

            // This exception will be caught by the catch statement below
            readFile2();

        } catch (FileNotFoundException ex) // catch the file not found exception
        {
            System.out.println("Found not file");
        }
        finally //will always come here in regardless if the exception occurs a not
        {
            System.out.println("Finally");
        }
    }

    public static void readFile2() throws FileNotFoundException {

        //we can create an exception object and throw it!
        throw new FileNotFoundException("My exception");
    }
}
```

```
File not found
Finally
```

Throwing Exception explicitly

Multiple catch exceptions

- A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

```
try
{
   //Protected code
}catch(ExceptionType1 e1)
{
   //Catch block
}catch(ExceptionType2 e2)
{
   //Catch block
}catch(ExceptionType3 e3)
{
   //Catch block
}
```

- You can have any number of catch after a single try.

Multiple catch exceptions

- When an exception occurs in the protected code, the exception is thrown to the first catch block in the list.
- If the data type of the exception thrown matches ExceptionType1, it gets caught there.

```
try
{
   //Protected code
}catch(ExceptionType1 e1)
{
   //Catch block
}catch(ExceptionType2 e2)
{
   //Catch block
}catch(ExceptionType3 e3)
{
   //Catch block
}
```

Multiple catch exceptions

- If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

```
try
{
   //Protected code
}catch(ExceptionType1 e1)
{
   //Catch block
}catch(ExceptionType2 e2)
{
   //Catch block
}catch(ExceptionType3 e3)
{
   //Catch block
}
```

Multiple catch exceptions

```
int[] nums = {2,3,4};
try{
    //might throw FileNotFoundException here
    Scanner fileScanner = new Scanner(new File("test.txt"));
    String data = fileScanner.nextLine();
    //might throw NumberFormatException here
    int index = Integer.parseInt(data);
    //might throw ArrayIndexOutOfBoundException here
    System.out.println(nums[index]);
}catch(FileNotFoundException e){
    System.out.println("File not found");
}catch(NumberFormatException e){
    System.out.println("Data is not int");
}
```

Note: ArrayIndexOutOfBoundException is an unchecked exception and need not be caught.

Multiple catch exceptions

- We can have a general (Parent) exception at the end to catch all the other exceptions.

```
int[] nums = {2,3,4};
try{
    //might throw FileNotFoundException here
    Scanner fileScanner = new Scanner(new File("test.txt"));
    String data = fileScanner.nextLine();
    //might throw NumberFormatException here
    int index = Integer.parseInt(data);
    //might throw ArrayIndexOutOfBoundException here
    System.out.println(nums[index]);
}catch(FileNotFoundException e){
    System.out.println("File not found");
}catch(NumberFormatException e){
    System.out.println("Data is not int");
}catch(Exception e){
    System.out.println("Exception in codes");
}
```

This will catch the ArrayIndexOutOfBound Exception

Multiple catch exceptions Rules

- ExceptionType1 cannot be a parent class of ExceptionType 2 and ExceptionType3
- ExceptionType2 cannot be a parent class of ExceptionType 3.

```
try
{
   //Protected code
}catch(ExceptionType1 e1)
{
   //Catch block
}catch(ExceptionType2 e2)
{
   //Catch block
}catch(ExceptionType3 e3)
{
   //Catch block
}
```

Multiple catch exceptions Rules

```
int[] nums = {2,3,4};
try{
    //might throw FileNotFoundException here
    Scanner fileScanner = new Scanner(new File("test.txt"));
    String data = fileScanner.nextLine();
    //might throw NumberFormatException here
    int index = Integer.parseInt(data);
    //might throw ArrayIndexOutOfBoundException here
    System.out.println(nums[index]);
}catch(Exception e){
    System.out.println("Exception in codes");
}
catch(FileNotFoundException e){
    System.out.println("File not found");
}catch(NumberFormatException e){
    System.out.println("Data is not int");
}
```

Unreachable catch block for FileNotFoundException. It is already handled by the catch block for Exception

2 quick fixes available:

Catch exceptions Rules

- The checked exception must happens somewhere in the codes

```
int[] nums = {2,3,4};
try{
    Scanner scanner = new Scanner(System.in);
    String data = scanner.nextLine();
    //might throw NumberFormatException here
    int index = Integer.parseInt(data);
    //might throw ArrayIndexOutOfBoundException here
    System.out.println(nums[index]);
}catch(FileNotFoundException e){
    Sys
}catch(
    Sys
```

Unreachable catch block for FileNotFoundException. This exception is never thrown from the try statement body

2 quick fixes available:

Catch exceptions Rules

- For unchecked exception, it is ok to catch it even it will never happen.

```
public static void test(){

    try
    {
        badMethod();
        System.out.print("A");
    }
    catch (ArrayIndexOutOfBoundsException ex)
    {
        System.out.print("B");
    }
    finally
    {
        System.out.print("C");
    }

}

public static void badMethod() {}
```

Troubleshooting Exception

- Use the error messages provided by Exception object.

public String getMessage()
Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.

public String toString()
Returns the name of the class concatenated with the result of getMessage()

public void printStackTrace()
Prints the result of toString() along with the stack trace to System.err, the error output stream.

Troubleshooting Exception

- Use getMessage()

```
try{
    Scanner file = new Scanner(new File("data.dat"));
}catch(FileNotFoundException e){
    System.out.println(e.getMessage());
}
```

```
data.dat (No such file or directory)
```

Troubleshooting Exception

- Using toString()

```
try{
    Scanner file = new Scanner(new File("data.dat"));
}catch(FileNotFoundException e){
    System.out.println(e);
}
```

```
java.io.FileNotFoundException: data.dat (No such file or directory)
```

Troubleshooting Exception

- Use printStackTrace()

```
try{

    Scanner file = new Scanner(new File("data.dat"));

}catch(FileNotFoundException e){

    e.printStackTrace();
}
```

```
java.io.FileNotFoundException: data.dat (No such file or directory)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(FileInputStream.java:120)
        at java.util.Scanner.<init>(Scanner.java:636)
        at FileNotFoundExample.main(FileNotFoundExample.java:11)
```

Troubleshooting Exception

- Only use the default system error messages during development.
- For production codes, NEVER display the system error messages.
- Your user will not be able to understand those error messages.

Creating your own Exception

- We can create exceptions class in Java
- All exceptions must be a child of Throwable.
- To create checked exception extend the Exception class.
- To create unchecked exception, extend the RuntimeException class.
- Not a practice to extend Error class in development.

Creating your own Exception

```
public class InvalidFruitException extends Exception{

    public InvalidFruitException(String fruit){
        super(fruit + " is invalid");
    }

}
```

Customized error message.
This message can be retrieved using the getMessage() method when the exception is thrown.

Creating your own Exception

```
import java.util.*;

public class FruitBasket {

    ArrayList<String> validFruit;
    ArrayList<String> basket;

    public FruitBasket(){
        basket = new ArrayList<String>();
        validFruit = new ArrayList<String>();
        validFruit.add("Apple");
        validFruit.add("Pear");
        validFruit.add("Orange");
    }

    public void addFruit(String fruit) throws InvalidFruitException {

        if(!validFruit.contains(fruit)){
            throw new InvalidFruitException(fruit);
        }

        basket.add(fruit);
    }

}
```

This fruit basket can only
Take in Apple, Pear and Orange

Creating your own Exception

```
FruitBasket basket = new FruitBasket();

try{
    basket.addFruit("Apple");
    System.out.println("Apple added");
    basket.addFruit("Durian");
    System.out.println("Durian added");
    basket.addFruit("Orange");
    System.out.println("Orange added");

}catch(InvalidFruitException e){
    System.out.println(e.getMessage());
}
```

```
Apple added
Durian is invalid
```

Keyboard class

- Extensive use of catching exception.

```
public static int readInt(String prompt) {
  int input = 0;
  boolean valid = false;
  while (!valid) {
    try {
      input = Integer.parseInt(readString(prompt));
      valid = true;
    } catch (NumberFormatException e) {
      System.out.println("*** Please enter an integer ***");
    }
  }
  return input;
}
```

Assertion

- Use assertions primarily for debugging and identifying logic errors in an application.
- Must explicitly enable assertions when executing a program
  - Assertions reduce performance.
  - Assertions are unnecessary for the program's use at production environment.

Assertion

- Assertion should not be used to replace exception handling.
- Exception handling deals with unusual circumstances during program execution.
- Assertions are to assure the correctness of the program.
- Exception handling addresses robustness and assertion addresses correctness.
- Like exception handling, assertions are not used for normal tests, but for internal consistency and validity checks.
- Assertions are checked at runtime and can be turned on or off at startup time.

Assertion

- Assertion example in Lab 1

```java
int score = 0;

// function 1
try {
    double[] numbers = { 0, 1, 2, 3 };
    double n = Lab1.largest(numbers);
    //throws AssertionError exception if
    //the condition n==3 is false
    assert n == 3;
    score += 1;
    System.out.println("Function 1 passed");
} catch (AssertionError ae) {
    System.out.println("Function 1 failed");
}
```