CL309 - Object Oriented Analysis and Design Lab
Lab#08

Object-Oriented Programming: Exception Handling

Objectives

Understanding the concepts of

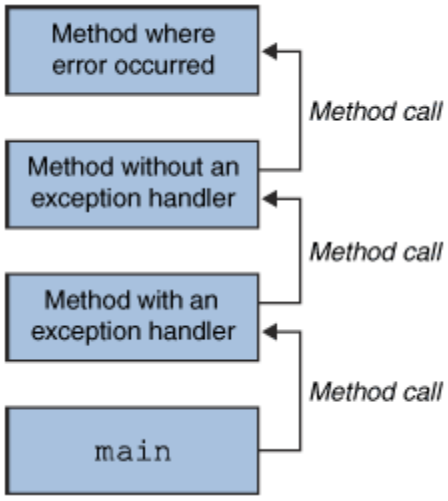
- Exception Handling
- Final Keyword

7.1. What Is an 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.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called *throwing an exception*.

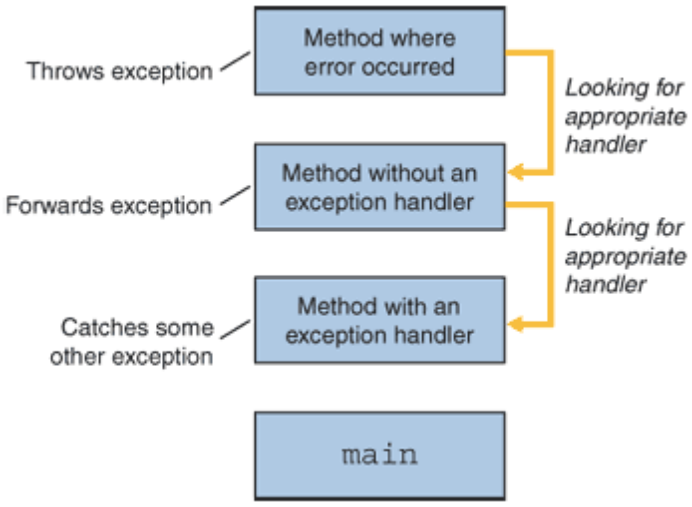
After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the *call stack*.



The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an *exception handler*. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the

runtime system passes the exception to the handler. An *exception handler* is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

The exception handler chosen is said to *catch the exception*. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the Figure below, the runtime system (and, consequently, the program) terminates.



7.2. The Catch or Specify Requirement

Valid Java programming language code must honor the *Catch* or *Specify* Requirement. This means that code that might throw certain exceptions must be enclosed by either of the following:

- A *try* statement that catches the exception. The try must provide a handler for the exception.
- A *method* that specifies that it can throw the exception. The method must provide a *throws* clause that lists the exception.

Code that fails to honor the Catch or Specify Requirement will not compile. Not all exceptions are subject to the Catch or Specify Requirement. To understand why, we need to look at the three basic categories of exceptions, only one of which is subject to the Requirement.

7.3. The Three Kinds of Exceptions

The first kind of exception is the ***checked exception***. These are exceptional conditions that a well-written application should anticipate and recover from. For example, suppose an application prompts a user for an input file name, then opens the file by passing the name to the constructor for *java.io.FileReader*. Normally, the user provides the name of an existing, readable file, so the construction of the FileReader object succeeds, and the execution of the application proceeds normally. But sometimes the user supplies the name of a nonexistent file, and the constructor throws java.io.FileNotFoundException. A well-written program will catch this exception and notify the user of the mistake, possibly prompting for a corrected file name. Checked exceptions are subject to the Catch or Specify Requirement. All exceptions are checked exceptions, except for those indicated by Error, RuntimeException, and their subclasses.

The second kind of exception is the ***error***. These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from. For example, suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction. The unsuccessful read will throw *java.io.IOError*. An application might choose to catch this exception, in order to notify the user of the problem — but it also might make sense for the program to print a stack trace and exit. Errors are not subject to the Catch or Specify Requirement. Errors are those exceptions indicated by Error and its subclasses.

The third kind of exception is the ***runtime exception***. These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from. These usually indicate programming bugs, such as logic errors or improper use of an API. For example, consider the application described previously that passes a file name to the constructor for *FileReader*. If a logic error causes a null to be passed to the constructor, the constructor will throw *NullPointerException*. The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur.

Runtime exceptions are not subject to the Catch or Specify Requirement. Runtime exceptions are those indicated by RuntimeException and its subclasses.

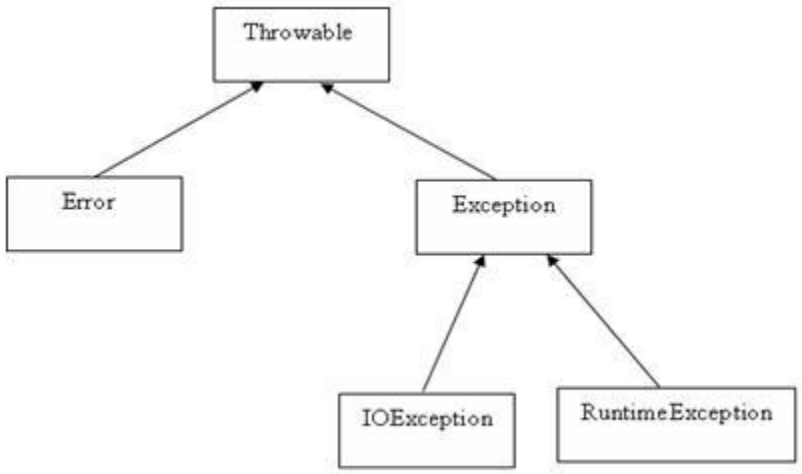
Errors and runtime exceptions are collectively known as *unchecked exceptions.*

7.4. Exception Hierarchy

All exception classes are subtypes of the *java.lang.Exception class*. The exception class is a subclass of the *Throwable* class. Other than the exception class there is another subclass called *Error* which is derived from the *Throwable* class.

Errors are not normally trapped form the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of Memory. Normally programs cannot recover from errors.

The Exception class has two main subclasses : IOException class and RuntimeException Class.



7.5. Exceptions Methods

Following is the list of important methods available in the Throwable class.

SN	Methods with Description
1	**public String getMessage()** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	**public Throwable getCause()** Returns the cause of the exception as represented by a Throwable object.
3	**public String toString()** Returns the name of the class concatenated with the result of getMessage()
4	**public void printStackTrace()** Prints the result of toString() along with the stack trace to System.err, the error output stream.
5	**public StackTraceElement [] getStackTrace()** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	**public Throwable fillInStackTrace()** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

7.6. Catching Exceptions

A method catches an exception using a combination of the **try** and **catch** keywords. 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
}
```

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example:

The following is an array is declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest{

   public static void main(String args[]){
   int a[] = new int[2];
      try{
```

```
         System.out.println("Access element three :" + a[3]);
      }catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Exception thrown  :" + e);
      }
      System.out.println("Out of the block");
   }
}
```

This would produce following result:

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

7.7. Multiple catch Blocks

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
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If 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. 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.

Example:

Here is code segment showing how to use multiple try/catch statements.

```
Try
{
   file = new FileInputStream(5ilename);
   x = (byte) file.read();
}catch(IOException i)
{
   i.printStackTrace();
   return -1;
}catch(FileNotFoundException f) //Not valid!
{
   f.printStackTrace();
```

```
   return -1;
}
```

7.8. The throws/throw Keywords

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.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword. **Try to understand the difference in throws and throw keywords.**

The following method declares that it throws a RemoteException:

```
import java.io.*;
public class className
{
   public void deposit(double amount) throws RemoteException
   {
      // Method implementation
      throw new RemoteException();
   }
   //Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a *RemoteException* and an *InsufficientFundsException*:

```
import java.io.*;
public class className
{
   public void withdraw(double amount) throws RemoteException,
                              InsufficientFundsException
   {
       // Method implementation
   }
   //Remainder of class definition
}
```

In some cases while developing our own applications, we need to specify custom types of exceptions to handle custom errors that may occur during program execution. A custom exception is a class that inherits Exception class or any of its subclasses, we will define it as using extends Exception in class declaration.

```
class MyCustomException extends Exception{
       private String message;
       public MyCustomException(String message){
              this.message = message;
```

```
		}

		public String toString(){
			return message;
		}
	}
```

To use your custom exception, declare an object of it and throw that object using throw keyword. It is optional to declare the method containing throw statement with throws keyword. In the following example, the program reads student id, this id should be of length 7 and consists only of digits, otherwise it throws an exception.

```
class InvalidIDException extends Exception{
		private String message;
		public InvalidIDException(String message){
		this.message = message;
	}

	public String toString(){
		return message;
	}
}

Import javax.swing.*;
class StudentsData{
public static void main(String[] args){
	String id, name;
	name = JOptionPane.showInputDialog("Enter student name");
	id = JOptionPane.showInputDialog("Enter student ID");
	try{
		verfyID(id);
	}

	catch(InvalidIDException e){
		JoptionPane.showMessageDialog(null, e.toString());
	}

	public static void verifyID(String id) throws InvalidIDException{
		if(id.length() != 7){
		throw new InvalidIDException("Check ID length");
		}
		try{
			Long.parseLong(id);
		}
		catch(Exception err){
			throw new InvalidIDException("ID can contain only digits");
		}
	}
}
```

7.8.1. The finally Keyword

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

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax:

```
try
```

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

Example:

```
public class ExcepTest{

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

This would produce following result:

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

Note the followings:

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses whenever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

7.8.2. Declaring you own Exception

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below:

```
class MyException extends Exception{
}
```

You just need to extend the Exception class to create your own Exception class. These are considered to be checked exceptions. The following InsufficientFundsException class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

Example:

```
// File Name InsufficientFundsException.java
import java.io.*;

public class InsufficientFundsException extends Exception
{
   private double amount;
   public InsufficientFundsException(double amount)
   {
      this.amount = amount;
   }
   public double getAmount()
   {
      return amount;
   }
}
```

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

```
// File Name CheckingAccount.java
import java.io.*;

public class CheckingAccount
{
   private double balance;
   private int number;
   public CheckingAccount(int number)
   {
      this.number = number;
   }
   public void deposit(double amount)
   {
      balance += amount;
   }
   public void withdraw(double amount) throws
                               InsufficientFundsException
   {
      if(amount <= balance)
      {
         balance -= amount;
      }
      else
      {
         double needs = amount - balance;
```

```
         throw new InsufficientFundsException(needs);
      }
   }
   public double getBalance()
   {
      return balance;
   }
   public int getNumber()
   {
      return number;
   }
}
```

The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

```
// File Name BankDemo.java
public class BankDemo
{
   public static void main(String [] args)
   {
      CheckingAccount c = new CheckingAccount(101);
      System.out.println("Depositing $500...");
      c.deposit(500.00);
      try
      {
         System.out.println("\nWithdrawing $100...");
         c.withdraw(100.00);
         System.out.println("\nWithdrawing $600...");
         c.withdraw(600.00);
      }catch(InsufficientFundsException e)
      {
         System.out.println("Sorry, but you are short $"
                                 + e.getAmount());
         e.printStackTrace();
      }
    }
}
```

Compile all the above three files and run BankDemo, this would produce following result:

```
Depositing $500...

Withdrawing $100...

Withdrawing $600...
Sorry, but you are short $200.0
InsufficientFundsException
        at CheckingAccount.withdraw(CheckingAccount.java:25)
        at BankDemo.main(BankDemo.java:13)
```

7.8.3. Common Exceptions

- A list of common Java Exceptions can be found here http://rymden.nu/exceptions.html

Exercises

1. Write a program that count how many prime numbers between minimum and maximum values provided by user. If minimum value is greater than or equal to maximum value, the program should throw a *InvalidRange* exception and handle it to display a message to the user on the following format:Invalid range: minimum is greater than or equal to maximum. For example, if the user provided 10 as maximum and 20 as minimum, the message should be: Invalid range: 20 is greater than or equal to 10.