Description

This week we will get an overview of exceptions and exception handling. We will learn to declare exceptions in a method header, throw exceptions, using try-catch block to handle exceptions and use finally clause.

Introduction

Runtime errors are very common while a program is running. The program breaks at runtime if the environment detects an operation that is impossible to carry out. For example,

1. Accessing an array using an index out of bounds, program will get a runtime error with an ArrayIndexOutOfBoundsException.
2. We need to create a Scanner object using new Scanner(new File(filename)), to read data from a file. If the file does not exist, the program will get a runtime error with a FileNotFoundException.

In Java, runtime errors are caused by exceptions. **An exception** is an object that represents an error or a condition that prevents execution from preceding normally. If the exception is not handled, the program will terminate abnormally.

Exception Handling

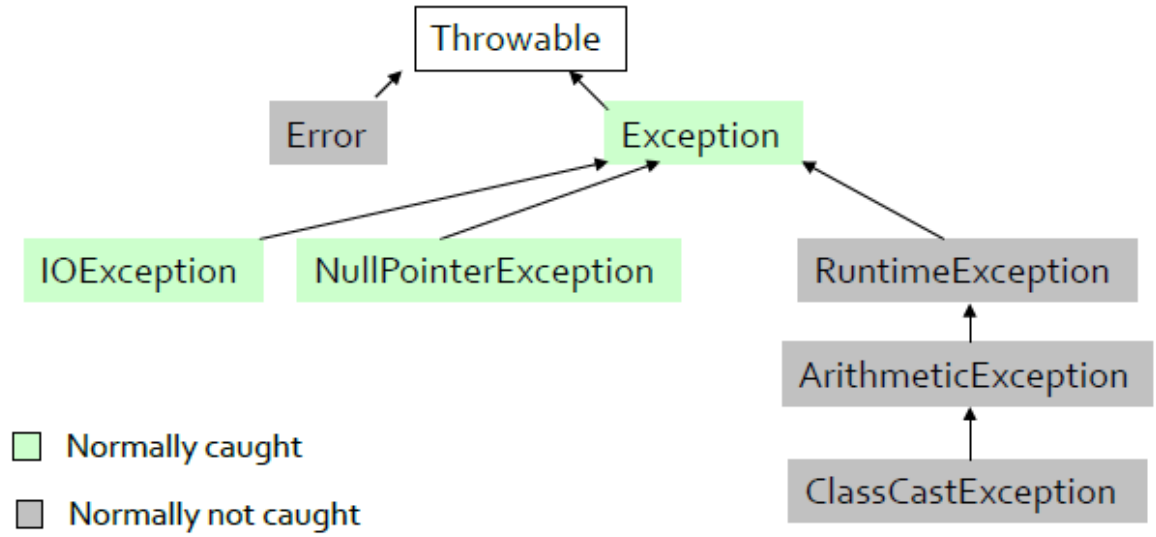
Exception Handling is the process of handling the exceptions so that the program can continue to run or terminate gracefully. Exceptions must be caught in try and catch block.

```
try{
    doSomething();
    }catch(ArrayIndexOutOfBoundsException e){
          e.printStackTrace();   }
```

```
Scanner scan = new Scanner(System.in);
boolean flag = true;
do {
    try {
        System.out.print("Enter an integer:");
        int number = scan.nextInt();
        System.out.println("The number entered is " + number);
        flag = false;
    } catch (InputMismatchException ex) {
         System.out.println("Try again, an integer is required");
         scan.next();
    }
} while (flag);
```

When executing scan.nextInt(), an InputMismatchException occurs if the input entered is not an integer. Suppose 1.6 is entered and InputMismatchException occurs. The control is transferred to the catch block. The statements in the catch block are now executed. The variable flag controls the loop (changes to false only when valid input is received) and scan.next() discards the current input line so that user can enter a new line of input.

Exception Types (Java Exception Classes)
There are many predefined exception classes in the Java API.



Some of the pre-defined exception classes in Java

Throwable

The superclass of all errors and exceptions. All Java exception classes inherit directly or indirectly from Throwable. Rarely used on its own.

Throwable Methods:

1. Throwable fillInStackTrace() – creates stack trace
2. String getMessage() – returns the message string
3. void printStackTrace() – prints out the stack trace
4. String toString() – a short description, usually the class type and message.

Error

Error is the superclass of all errors

- Most errors are abnormal conditions that a program should not try to catch.
- An error is an *unchecked exception*, which does not have to be caught or declared in a throws clause.
- Example Errors: AssertionError, AWTError, LinkageError, VirtualMachineError

Exception

This is the superclass of all exceptions but rarely used on its own.
Two groups of classes are derived from it.

- **The checked exceptions**–must be caught and / or declared in a throws clause.
- **The unchecked exceptions**–do not need be caught or declared in a throws clause.

RuntimeException, Error and their subclasses are known as unchecked exceptions. All other exceptions are known as checked exceptions, means the compiler forces the programmer to check and deal with them.

Usually, unchecked exceptions are programming logic errors. For ex., NullPointerException is thrown if we access an object through a reference variable before an object is assigned to it, an IndexOutOfBoundsException is thrown if we access an element in an array which is outside of array bounds. These are logic errors and must be corrected in the program.

Note: To avoid cumbersome overuse of try-catch blocks, Java does not mandate to catch or declare unchecked exceptions.

More on Exception Handling

Java exception handling is based on three operations: declaring an exception, throwing an exception and catching an exception.

1. Declaring exceptions: Every method must state the types of checked exceptions it might throw in the method header, so that the caller of the method is informed of the exception. To declare an exception in a method, use the **throws** keyword in the method header.
   public void myMethod() throws IOException

   A method can throw multiple exceptions, separated by commas after throws:
   public void myMethod() throws Exception1, Exception2, …, Exception N

2. Throwing an exception: You throw an exception using the ***throw*** statement. You can create an instance of an appropriate exception type and throw it. Your method must declare any checked exception it will throw.

```
public double findSqrt(double d)
            throws IllegalArgumentException{
    if(d < 0.0){
        throw new IllegalArgumentException(
         "Negative value: " + d);
    }
}
```

The throws clause:

- When a method throws a checked exception, it must declare this in a ***throws*** clause.

- Multiple exceptions are listed in a throws clause separated by commas.
- If a method invokes another method which throws exceptions it must catch them, or declare them in its throws clause

3. Catching exceptions: When an exception is thrown, it can be caught and handled in a ***try-catch*** block.

```
try{
    double sq = findSqrt(35.4);
}catch(IllegalArgumentException e){
        e.printStackTrace();
}
```

This will catch any IllegalArgumentException thrown from the try block.

Catching multiple exceptions: If the code in a try block throws multiple exceptions, you need multiple catch statements to handle them.

```
try{
     int result = 1 / 0;
     double sq = findSqrt(35.4);
  }catch(ArithmeticException e) {
      e. printStackTrace();
 }catch(IllegalArgumentException ex){
      ex.printStackTrace();
```

If no exception arises during the execution of the try block, the catch blocks are skipped.

If one of the statements inside the try block throws an exception, JVM skips the remaining statements in the block and starts examining each catch block in turn, from first to last, to see whether the type of the exception object is an instance of the exception class in the catch block. If so, the code in the catch block is executed.

If no handler/catch is found, the program terminates and prints an error message on the console.

Catching an Exception superclass: Since the exception classes are arranged into a hierarchy, catching a superclass will catch all subclasses of that exception.

```
try{
     int result = 1 / 0;
     double sq = findSqrt(35.4);
  }catch(Exception e) {
      e. printStackTrace();
 }
```

Example: Declaring, throwing and catching exception

I have modified setBalance method in the BankAccount class to throw an exception if the balance is negative.

```
public void setBalance(double balance)throws IllegalArgumentException {
 if(balance >= 0)
 this.balance = balance;
 else
 throw new IllegalArgumentException("Balance cannot be negative");
 }
```

```
public static void main(String[] args) {
      BankAccount account1 = null;
      try{
          BankAccount account2 = new BankAccount(2000);
          account1 = new BankAccount(-1000);
          System.out.println(account1.toString() + "\n");
          System.out.println(account2.toString());
        } catch(IllegalArgumentException e){
              System.out.println(e);
      }
```

Here is the output:

```
java.lang.IllegalArgumentException: Balance cannot be negative
```

Declaring your own exceptions

If there is no built-in exception that meets your needs, you can define your own

- Most exceptions you define are derived from Exception or from a subclass of Exception, such as IOException.
- Often the name of the exception and the message is all you need.
- In other cases, you might want to include additional information with the exception to help identify the problem.

Let's create custom exception class to handle negative balance in BankAccount.

```
public class InvalidInputException extends Exception{
  private double num;
  public InvalidInputException(double num){
       super("Invalid Balance " + num);
       this.num = num;
}  }
```

This custom exception class extends java.lang.Exception. The Exception class extends java.lang.Throwable. All the methods (e.g. getMessage(), toString(), and printStackTrace()) in Exception are inherited from Throwable. The Exception class contains four constructors. Two of them are often used:

Exception(): Constructs an exception with no message.

Exception(String message): Constructs an exception with the specified message.

```
public static void main(String[] args) {
BankAccount account1 = null;
try{
 account1 = new BankAccount(-1000);
System.out.println(account.toString());
 }catch(InvalidInputException e){
 System.out.println(e);
 }
```

Here is the output:

```
InvalidInputException: Invalid Balance -1000.0
```

The finally clause

Sometimes we need to do something regardless of whether an exception was thrown or not. This can be accomplished by including a ***finally*** clause at the end of a try block.

The code in the finally block will be executed

- At the end of the block if no exception was thrown
- At the end of a catch clause that handled an exception
- Just before an uncaught exception is propagated

```
try{
    tryStatements;
}catch(Exception ex){.....}
finally{
     finalStatements;
}
```

The code in the finally block, referred as finalStatements, is executed under all circumstances, regardless of whether an exception occurs in the try block or is caught. Consider three possibilities:

1. If no exception arises in the try block, finalStatements is executed.
2. If a statement causes an exception in the try block which is caught in the catch block, the rest of the statements in the try block are skipped, the catch block is executed and finalStatements are executed.

3. If one of the statements causes an exception that is not caught in any catch block, the other statements in the try block are skipped, finalStatements are executed and the exception is passed to the caller of this method.

```
java.io.PrintWriter output = null;
 try{
 output = new java.io.PrintWriter("text.txt");
 output.println("Welcome to Java");
 }catch(java.io.IOException ex){
 System.out.println(ex.toString());
 }finally{
 if(output != null)
 output.close();
 }
```

The following statements

output = new java.io.PrintWriter("text.txt");
output.println("Welcome to Java");

may throw an IOException, so they are placed inside try block. The statement output.close() closes the PrintWriter object output in the finally block. This statement is executed regardless of whether an exception occurs in the try block or is caught.