Exception Handling

Abdul Haseeb

Exception Fundamentals

- Exception:
  - Abnormal condition in a code sequence, which arises at run-time
  - Run Time Error
  - Disrupts the normal flow of program

Exception Fundamentals

- Some programming languages doesn’t provide exception handling
  - Manually handle
- Java provides exception handling

Exception Fundamentals

- Java Exception:
  - Object
    - Describes error/exception, which has occurred in the code.

- What happens when an exception arises?
  - Object related to that exception is created
  - Exception is caught at some point

Exception Fundamentals

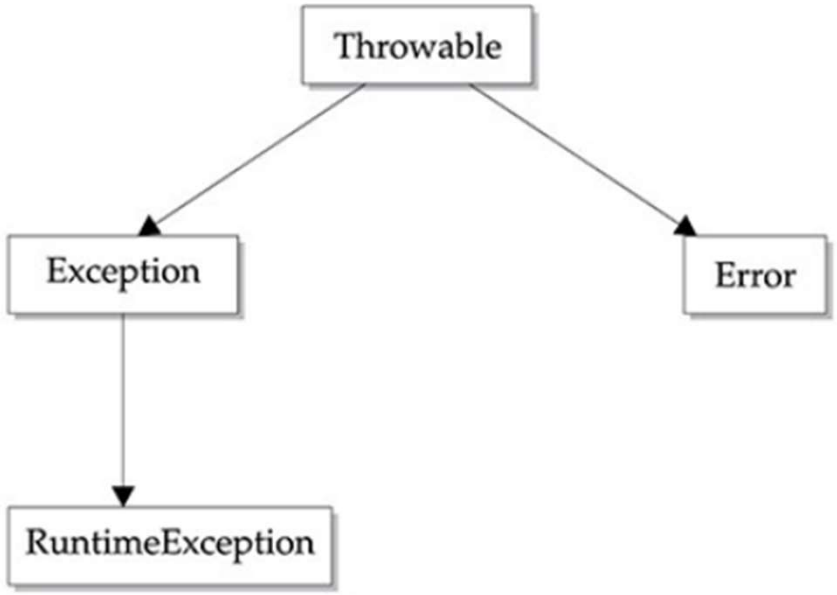
- Java Exception:
  - Exception can be generated by your code
    - Represent an error in your code
  - Can also be generated by JVM:
    - When you violate the rules of java execution environment
    - ArrayIndex out of Bound

Keywords

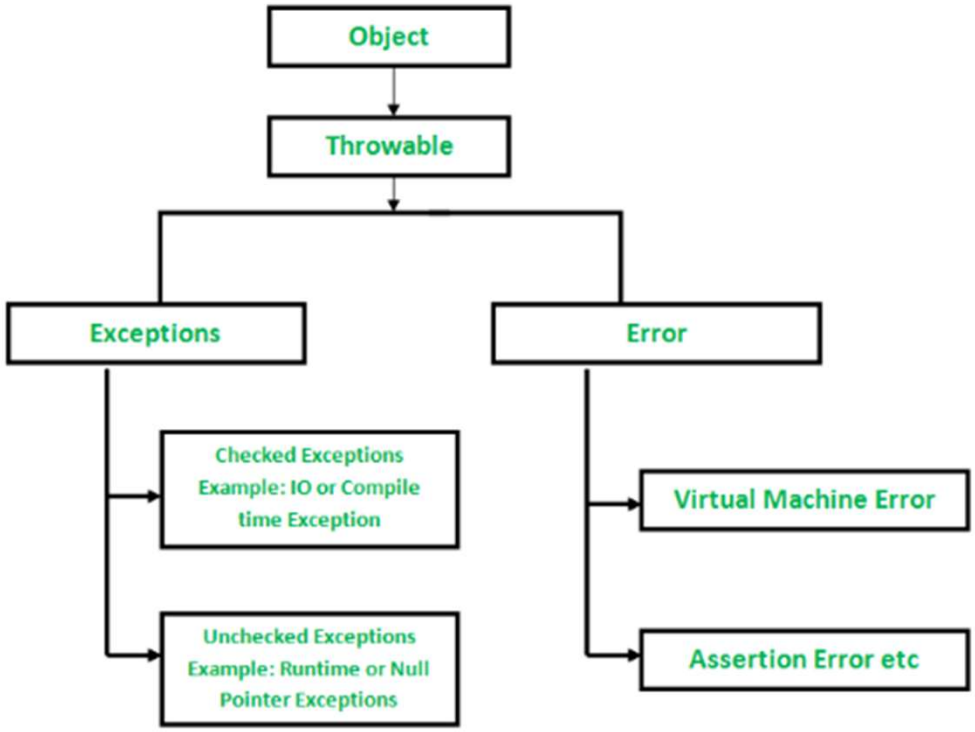
- try:
  - Contains code to be checked for an exception, and is thrown
- catch:
  - thrown Exception is caught and handled
- throw:
  - Manually throw an exception
- throws:
  - If exception is thrown, outside of the method, use throws
- finally:
  - After try block completes, any block that must be executed after it, is placed in finally 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
}
```

Hierarchy



Hierarchy


Object
Throwable
Exceptions
Error
Checked Exceptions
Example: IO or Compile
time Exception
Unchecked Exceptions
Example: Runtime or Null
Pointer Exceptions
Virtual Machine Error
Assertion Error etc

Checked vs Unchecked Exception

- Checked Exceptions occur frequently and JVM is very careful:
  - IO, Interrupt and SQLException
- Unchecked Exceptions rarely, JVM not that much careful:
  - Run Time Exception like: Arithmetic, Null Pointer Exception etc
  - Error: Stack Overflow etc

Exception vs Error

- Exception:
  - Type of conditions that user program must handle
- Error:
  - Exceptions that are not expected to be caught under normal circumstances
  - Indicate issues with java run-time environment, lack of resources:
    - Stack overflow
    - Memory crashes

Uncaught Exceptions

- Lets see what happens if we don’t provide exception handling code
- Lets attempt division by zero
- Caught by Run-Time

Stack Trace, Complete Record that from where the error Originated

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

Using Try-Catch

- Java Run-Time was handling exceptions
- But Handling Exception by yourself has advantages:
  - Prevents program from automatically terminating
  - It allows us to fix the error

Try this Code!

```
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("This will not be printed.");
    } catch (ArithmeticException e) { // catch divide-by-zero error
     System.out.println("Division by zero.");
    }

    System.out.println("After catch statement.");
  }
}
```

Another Example

```
// Handle an exception and move on.
import java.util.Random;

class HandleError {
  public static void main(String args[]) {
    int a=0, b=0, c=0;
    Random r = new Random();

    for(int i=0; i<32000; i++) {
      try {
        b = r.nextInt();
        c = r.nextInt();
        a = 12345 / (b/c);
      } catch (ArithmeticException e) {
        System.out.println("Division by zero.");
        a = 0; // set a to zero and continue
      }
      System.out.println("a: " + a);
    }
  }
}
```

Description of an exception

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

Multiple Catch Blocks

- In some cases more than one exception, caused by a single piece of code
  - Uses Multiple catch blocks,one for each kind of exception
  - After occurrence of exceptions:
    - Catch blocks are examined in sequence
    - After one executes, others are bypassed

Division by Zero, Array Index out of Bound example

- Take integer input from user, using that input divide the number, (Zero or non zero)
- Try to access an element outside the range

Multiple Catch Blocks

- When using multiple catch blocks:
  - Make sure that subclass exception must precede the superclass exception, otherwise it will create unreachable code (Compile Time Error)
  - Because a superclass is aware of its children

```
/* This program contains an error.

   A subclass must come before its superclass in
   a series of catch statements. If not,
   unreachable code will be created and a
   compile-time error will result.
*/
class SuperSubCatch {
  public static void main(String args[]) {
    try {
      int a = 0;
      int b = 42 / a;
     } catch(Exception e) {
       System.out.println("Generic Exception catch.");
    }
    /* This catch is never reached because
       ArithmeticException is a subclass of Exception. */
    catch(ArithmeticException e) { // ERROR - unreachable
      System.out.println("This is never reached.");
    }
  }
}
```

Throws

- If method is capable of causing an **exception/exceptions** that it doesn't handle
- It then tells that I may cause an **exception**
- It tells that with **throws** clause
- Callers of methods must guard themselves:
  - Put that method in **try** block while calling

Throws

- Create static method divide, accepting two arguments
- Use throws to show it may throw Arithmetic Exception

```java
public class TestThrows {
    //defining a method
    public static int divideNum(int m, int n) throws ArithmeticException {
        int div = m / n;
        return div;
    }
    //main method
    public static void main(String[] args) {
        TestThrows obj = new TestThrows();
        try {
            System.out.println(obj.divideNum(45, 0));
        }
        catch (ArithmeticException e){
            System.out.println("\nNumber cannot be divided by 0");
        }

        System.out.println("Rest of the code..");
    }
}
```

Throw

- Used to throw an exception explicitly by a programmer
- We throw an exception inside method, so that caller of method is shown with:
  - Which type of exception must be handled.

Throw

```
public static void checkNum(int num) {
    if (num < 1) {
        throw new ArithmeticException("\nNumber is negative, cannot calculate square");
    }
    else {
        System.out.println("Square of " + num + " is " + (num*num));
    }
```

Throw and Throws

Finally

- After a try catch block has completed execution, finally can be executed
- Finally will always execute even if exception isn’t caught
- Finally block is optional
- Try block requires at least one catch block
  - If we omit catch block then we should give a finally

Java’s Built in Exceptions

- Available in java.lang
- Check out from the book

Assignment

- Find out how can you create your own exceptions