JAVA EXCEPTIONS

Compiled By FadelK

Objectives

- Learn what an exception is
- Learn how to use a try/catch block to handle exceptions
- Become acquainted with the hierarchy of exception classes
- Learn about checked and unchecked exceptions
- Learn how to handle exceptions within a program

Try The Following Code

```
public class Exc1 {
 public static void main(String[] args) {
  String[] ss={"Fadel","Waheed"};
  System.out.println(ss[2]);
 }
}
```

Output

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 2
        at Exc1.main(Exc1.java:5)
Java Result: 1
```

```
public class Exc1 {
 public static void main(String[] args) {
  int x= 1;
  int y =0;
  System.out.println(x/y);
 }
}
```

Output

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Exc1.main(Exc1.java:6)
Java Result: 1
```

Introduction

- An exception is an abnormal event that arises during the execution of the program and disrupts the normal flow of the program.
- Abnormality do occur when your program is running. For example, you might expect the user to enter an integer, but receive a text string; or an unexpected I/O error pops up at runtime.
- What really matters is "what happens after an abnormality occurred?" In other words, "how the abnormal situations are handled by your program."
- If these exceptions are not handled properly, the program terminates abruptly and may cause severe consequences. For example, the network connections, database connections and files may remain opened; database and file records may be left in an inconsistent state.
- Java has a built-in mechanism for handling runtime errors, referred to as exception handling. This is to ensure that you can write robust programs for mission-critical applications.

Previous Practices

- Older programming languages such as C/C++ have some drawbacks in exception handing. For example, suppose the programmer wishes to open a file for processing:

1. The programmers are not made to aware of the exceptional conditions. For example, the file to be opened may not necessarily exist. The programmer therefore did not write codes to test whether the file exists before opening the file.
2. Suppose the programmer is aware of the exceptional conditions, he/she might decide to finish the main logic first, and write the exception handling codes later – this "later", unfortunately, usually never happens. In other words, you are not force to write the exception handling codes together with the main logic.
3. Suppose the programmer decided to write the exception handling codes, the exception handling codes interlinked with the main logic in many if-else statements. This makes main logic hard to follow and the entire program hard to read. For example:

```
if (file exists) {
  open file;
  while (there is more records to be processed) {
     if (no IO errors) {
        process the file record
     } else {
        handle the errors
     }
  }
  if (file is opened) close the file;
} else {report the file does not exist;}
```

Exceptions In Java

- Java overcomes these drawbacks by building the exception handling into the language rather than leaving it to the discretion of the programmers:

1. You will be informed of the exceptional conditions that may arise in calling a method. [Exceptions are declared in the method's signature.]
2. You are forced to handle exceptions while writing the main logic and cannot leave them as an afterthought. [Your program cannot compiled without the exception handling codes.]
3. Exception handling codes are separated from the main logic. [Via the try-catch-finally construct.]

Basic Try/Catch

```
import java.util.Scanner;
public class Exc1 {
 public static void main(String[] args) {
  Scanner s= new Scanner(System.in);
  System.out.println("Before the try catch block");
  try {
     System.out.println("Inside Try block before reading");
     int i = s.nextInt();
     System.out.println("Inside Try block after reading");
  } catch (Exception e) {
     System.out.println("Inside Catch Block");
  }
  System.out.println("After the try catch block");
 }
}
```

Output A

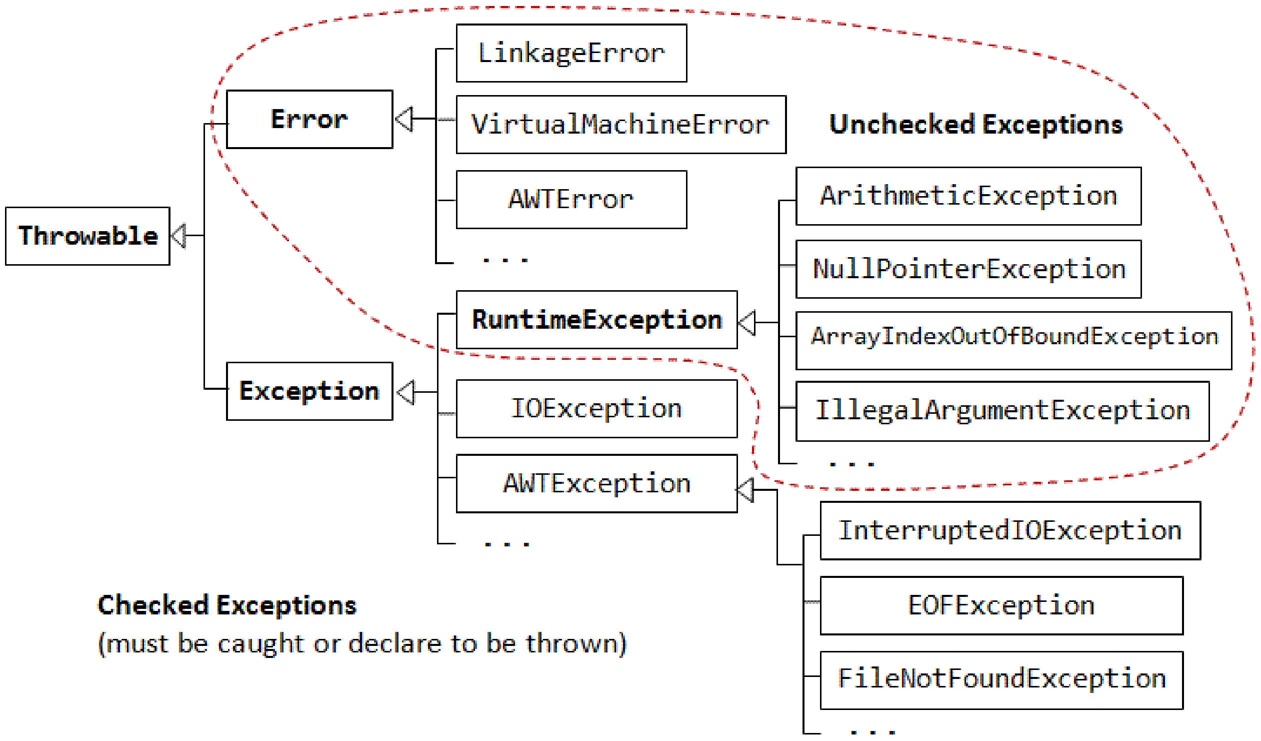
```
Before the try catch block
Inside Try block before reading
123
Inside Try block after reading
After the try catch block
```

Output B

```
Before the try catch block
Inside Try block before reading
rrrr
Inside Catch Block
After the try catch block
```

What is **catch (Exception e)**

- Exception Classes
  - The figure below shows the hierarchy of the Exception classes. The base class for all Exception objects is java.lang.Throwable, together with its two subclasses java.lang.Exception and java.lang.Error.



What is **catch (Exception e)** Cont.

- The Error class describes internal system errors (e.g., VirtualMachineError, LinkageError) that rarely occur. If such an error occurs, there is little that you can do and the program will be terminated by the Java runtime.
- The Exception class describes the error caused by your program (e.g. FileNotFoundException, divid by zero, IOException). These errors could be caught and handled by your program (e.g., perform an alternate action or do a graceful exit by closing all the files, network and database connections).

Better Try/Catch

```
import java.util.*;
public class Exc1 {
 public static void main(String[] args) {
  Scanner s= new Scanner(System.in);
  try {
     System.out.println("Enter X value");
     int x = s.nextInt();
     System.out.println("Enter Y value");
     int y = s.nextInt();
     System.out.println("X / Y = " +(x/y));
  } catch (InputMismatchException ime) {
     System.out.println("Invalid Integer Values");
  } catch (ArithmeticException ae){
     System.out.println("Could Not Calculate");
  }
 }
}
```

Output A

```
Enter X value
4
Enter Y value
2
X / Y = 2
```

Output B

```
Enter X value
rrr
Invalid Integer Values
```

Output C

```
Enter X value
44
Enter Y value
0
Could Not Calculate
```

How to Catch Exceptions

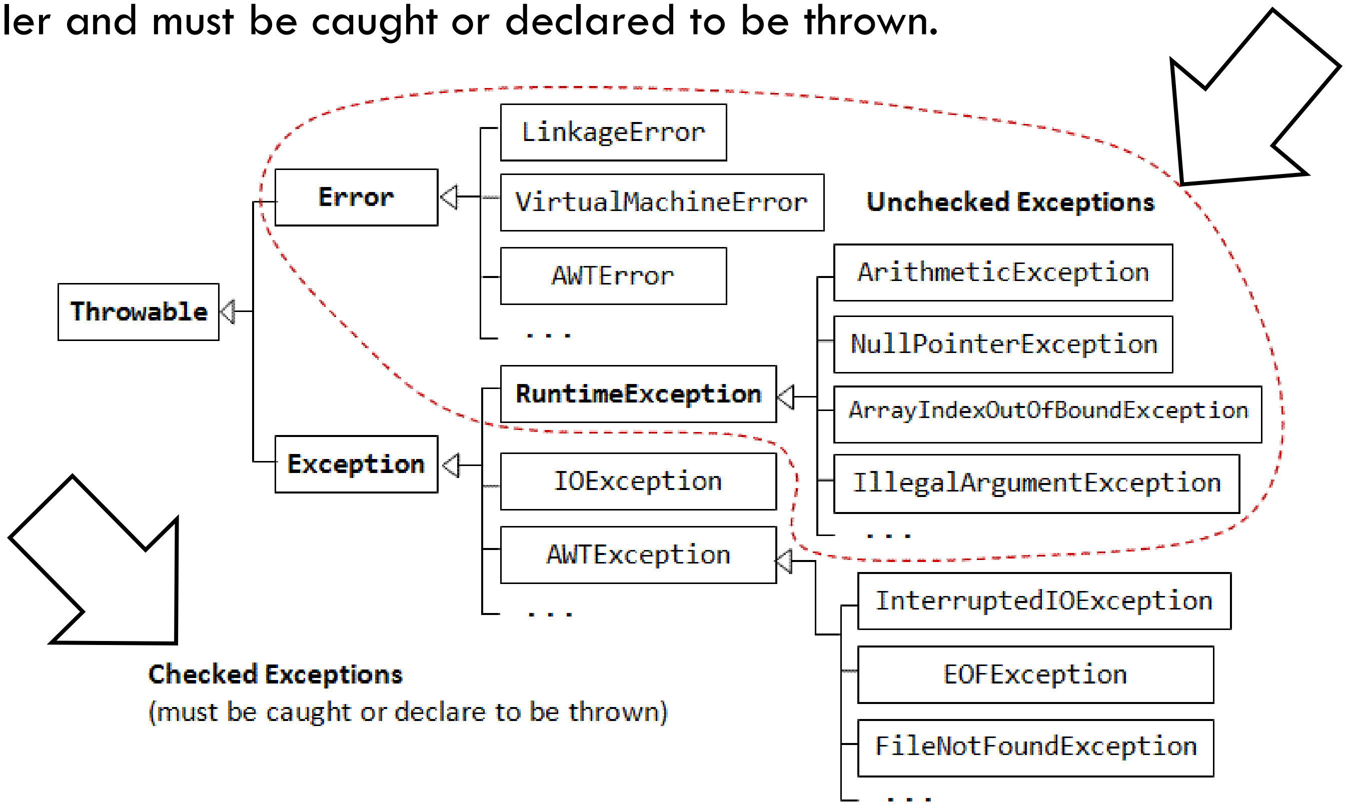
- Catch exceptions using try blocks:

```
try {
        // statements that might cause exceptions
        // possibly including function calls
} catch ( exception-1 id-1 ) {
        // statements to handle this exception
} catch ( exception-2 id-2 ) {
        // statements to handle this exception
    .
    .
    .
}
```

- Each catch clause specifies the type of one exception, and provides a name for it (similar to the way a function header specifies the type and name of a parameter). Java exceptions are objects, so the statements in a catch clause can refer to the thrown exception object using the specified name.

# Checked vs Unchecked Exceptions

- As illustrated, the subclasses of Error and RuntimeException are known as unchecked exceptions. These exceptions are not checked by the compiler, and hence, need not be caught or declared to be thrown in your program. This is because there is not much you can do with these exceptions.
- All the other exception are called checked exceptions. They are checked by the compiler and must be caught or declared to be thrown.



The Checked Exception

- Exceptions must be Declared
  - As an example, suppose that you want to use a java.util.Scanner to perform formatted input from a disk file. The signature of the Scanner's constructor with a File argument is given as follows:

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

  - The method's signature informs the programmers that an exceptional condition "file not found" may arise. By declaring the exceptions in the method's signature, programmers are made to aware of the exceptional conditions in using the method.

The Checked Exception

- Exceptions must be Handled
  - If a method declares an exception in its signature, you cannot use this method without handling the exception (you can't compile the program).
  - The program did not handle the exception declared, resulted in compilation error.

Source code that will never compile

```
import java.util.Scanner;
import java.io.File;
public class test {
    public static void main(String args[]){
      Scanner in = new Scanner(new File("test.in"));
   }
}
```

Source Code not to catch the exception but to throw it.

```
import java.util.Scanner;
import java.io.File;
public class test {
    public static void main(String args[]) throws Exception {
      Scanner in = new Scanner(new File("test.in"));
   }
}
```

The Checked Exception Cont.

Source Code that catches IO Exceptions.

```
import java.util.*;
import java.io.*;
public class Exc1 {
 public static void main(String[] args) {
  try {
     Scanner s= new Scanner(new File("test.in"));
     String str= s.next();
  } catch (IOException ioe) {
     System.out.println("Could not open the file");
  }
 }
}
```

What if I really don't care about the exceptions

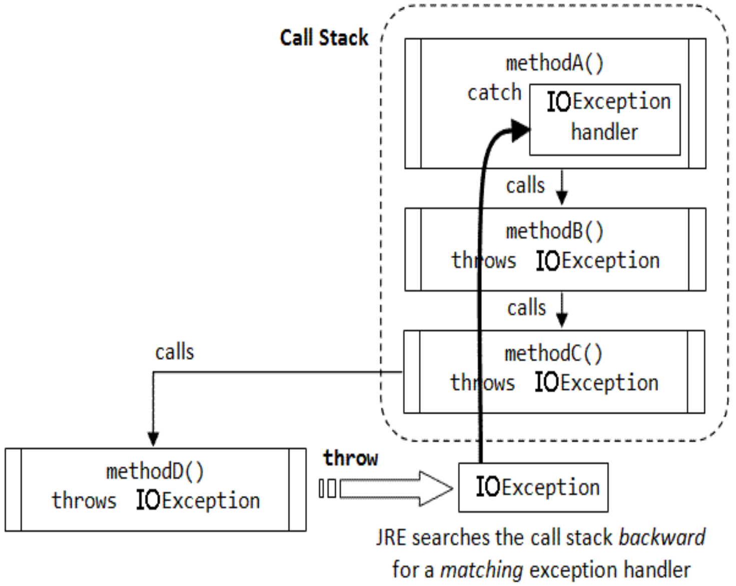
- Certainly not advisable other than writing toy programs. But to bypass the compilation error messages triggered by methods declaring unchecked exceptions, you could declare "throws Exception" in your main() (and other methods), as follows:

Source Code not to catch the exception but to throw it.

```
import java.util.Scanner;
import java.io.File;
public class test {
    public static void main(String args[]) throws Exception {
        Scanner in = new Scanner(new File("test.in"));
    }
}
```

Exception Handling Process

- When an exception occurs inside a Java method, the method creates an Exception object and passes the Exception object to the JRE (in Java term, the method "throws" an exception). The Exception object contains the type of the exception, and the state of the program when the exception occurs. The JRE is responsible for finding an exception handler to process the Exception object. It searches backward through the call stack until it finds a matching exception handler for that particular class of Exception object (in Java term, it is called "catch" the exception). If the JRE cannot find a matching exception handler in all the methods in the call stack, it terminates the program.
- This process is illustrated as follows. Suppose that methodD() encounters an abnormal condition and throws a IOException to the JRE. The JRE searches backward through the call stack for a matching exception handler. It finds methodA() having a IOException handler and passes the exception object to the handler. Notice that methodC() and methodB() are required to "throws IOException" upwards in order to compile the program.



How to Catch Exceptions

- Catch exceptions using try blocks:

```
try {
       // statements that might cause exceptions
       // possibly including function calls
} catch ( exception-1 id-1 ) {
       // statements to handle this exception
} catch ( exception-2 id-2 ) {
       // statements to handle this exception
    .
    .
    .
} finally {
       // statements to execute every time this try block
executes
}
```

- Notes:

1. Each catch clause specifies the type of one exception, and provides a name for it (similar to the way a function header specifies the type and name of a parameter). Java exceptions are objects, so the statements in a catch clause can refer to the thrown exception object using the specified name.
2. The finally clause is optional.
3. In general, there can be one or more catch clauses. If there is a finally clause, there can be zero catch clauses.

try-catch-finally

- The syntax of try-catch-finally is:

```
try {
   // main logic, uses methods that may throw Exceptions
   ...
} catch (Exception1 ex) {
   // error handler for Exception1
   ...
} catch (Exception2 ex) {
   // error handler for Exception1
   ...
} finally {   // finally is optional
   // clean up codes, always executed regardless of exceptions
   ...
}
```

- If no exception occurs during the running of the try-block, all the catch-blocks are skipped, and finally-block will be executed after the try-block. If one of the statements in the try-block throws an exception, the Java runtime ignores the rest of the statements in the try-block, and begins searching for a matching exception handler. It matches the exception type with each of the catch-blocks sequentially. If a catch-block catches that exception class or catches a superclass of that exception, the statement in that catch-block will be executed. The statements in the finally-block are then executed after that catch-block. The program continues into the next statement after the try-catch-finally, unless it is pre-maturely terminated or branch-out.
- If none of the catch-blocks matches, the exception will be passed up the call stack - if the method's signature declares that this checked exception to be thrown; or the exception is an unchecked exception. The current method terminates.

Notes

  - getMessage(): Returns the message specified if the object is constructed using constructor Throwable(String message).
  - toString(): Returns a short description of this Throwable object, consists of the name of the class, a colon ':', and a message from getMessage().
- A catch block catching a specific exception class can also catch its subclasses. Hence, catch(Exception ex) {...} catches all kinds of exceptions. However, this is not a good practice as the exception handler that is too general may unintentionally catches some subclasses' exceptions it does not intend to.
- The order of catch-blocks is important. A subclass must be caught (and placed in front) before its superclass. Otherwise, you receive a compilation error "exception XyzException has already been caught".
- The finally-block is meant for cleanup code such as closing the file, database connection regardless of whether the try block succeeds. The finally block is always executed (unless the catch-block pre-maturely terminated the current method).

Example

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
public class test {
   public static void main(String[] args) {
      try {         // main logic
         System.out.println("Start of the main logic");
         System.out.println("Try opening a file ...");
         Scanner in = new Scanner(new File("test.in"));
         System.out.println("File Found, processing the file ...");
         System.out.println("End of the main logic");
      } catch (FileNotFoundException e) {    // error handling separated from the main logic
         System.out.println("File Not Found caught ...");
      } finally {   // always run regardless of exception status
         System.out.println("finally-block runs regardless of the state of exception");
      }
      // after the try-catch-finally
      System.out.println("After try-catch-finally");
   }
}
```

Output when the FileNotFoundException triggered

```
Start of the main logic
Try opening a file ...
File Not Found caught ...
finally-block runs regardless of the state of
exception
After try-catch-finally
```

Output when no exception triggered:

```
Start of the main logic
Try opening a file ...
File Found, processing the file ...
End of the main logic
finally-block runs regardless of the state of
exception
After try-catch-finally
```

Notes

- A try-block must be accompanied by at least one catch-block or a finally-block.
- You can have multiple catch-blocks. Each catch-block catches only one type of exception.
- A catch block requires one argument, which is a throwable object (i.e., a subclass of java.lang.Throwable), as follows:
- catch (AThrowableSubClass aThrowableObject) {
-     // exception handling codes
- }
- You can use the following methods to inside the catch clause to retrieve the type of the exception and the state of the program from the Throwable object:
  - printStackTrace(): Prints this Throwable and its call stack trace to the standard error stream System.err. The first line of the outputs contains the result of toString(), and the remaining lines are the stack trace. This is the most common handler, if there is nothing better that you can do.