Exception Handling

An **exception** is any special condition that alters the normal flow of program execution. For example, you might try to divide by zero or open a file that does not exist. These errors can cause your program to terminate immediately. Java exception handling enables your Java applications to handle errors sensibly. Exception handling is a very important aspect of writing robust Java applications or components. When an error occurs in a Java program it usually results in an exception being thrown. How you throw, catch and handle these exception matters. Your perspective on errors depends on whether you are API **developer** or an API **client**.

• As a developer, your main concern is to **throw** the right exception at the right time. In most cases this is just a matter of putting in a throw statement and creating the right exception object.

• As a client, you may have to **handle** the exceptions thrown by methods you call. This is done with a try/ catch block. This prevents an error message from printing and the program from terminating.

By catching exceptions, you can recover from an unexpected error. For instance, if you try to open a file that doesn't exist or that you aren't allowed to read, Java will throw an exception. You can catch the exception, print an error message, and continue, instead of letting the program crash.

```
try {
   f = new FileInputStream("foo.txt");
   i = f.read();
}  catch (FileNotFoundException e1) {
   System.out.println(e1);
}  catch (IOException e2) {
   f.close();
}
```

This code does the following:

1. It executes the code inside the try braces.
2. If the try code executes normally, it skips over the catch clauses.
3. If the try code throws an exception, it jumps directly (without finishing the try code) to the first catch clause that matches the exception, and executes that catch clause. There is a **match** if the actual exception object thrown is the same class as, or a subclass of, the exception type listed in the catch clause. When the catch clause finishes executing, Java jumps to the next line of code immediately after all the catch clauses.

The code within a catch clause is called an **exception handler**.

If the `FileInputStream` constructor fails to open the file, it will throw a `FileNotFoundException`. The line "i = f.read()" is not executed; execution jumps directly to the exception handler.

`FileNotFoundException` is a subclass of `IOException`, so the exception matches both catch clauses. However, only one catch clause is executed— the first one that matches. The second catch clause would execute if the first were not present.

If the `FileInputStream` constructor runs without error, but read() throws an exception (e.g., because the disk is faulty), it typically generates some sort of `IOException` that isn't a `FileNotFoundException`. This causes the second catch clause to execute and close the file. Exception handlers are often used to recover from errors and clean up loose ends like open files.

Note that you don't need a catch clause for every exception that can occur. You can catch some exceptions and let others propagate.

Exceptions give us a way to escape a “sinking ship”. By letting the system throw an exception, or by throwing your own, you can move program execution out of a method whose purpose has been defeated. Throwing an exception is different from a return statement because

• You don't have to return anything.

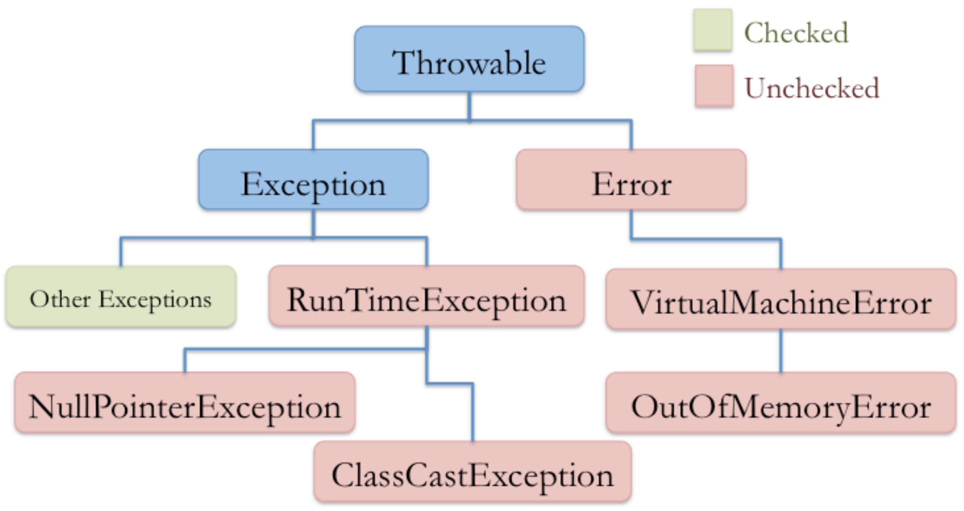
• An exception can fly several stack frames down the stack, not just one.

If an exception propagates all the way out of main() without being caught, the JVM prints an error message and halts.

An exception is itself a Java object belonging to the Exception class. This is subclass of a more general class of things you can throw and catch, which is called `Throwable`. Here's part of the `Throwable` hierarchy. The top-level class of things you can "throw" and "catch" is called `Throwable`.

There are checked and unchecked exceptions. Checked exceptions are those Exceptions for which we need to provide exception handling while unchecked exceptions are those for which providing exception handlers is optional.

Here’s part of the `Throwable` class hierarchy.



An Error generally represents a fatal error, like running out of memory or stack space. Although you can throw or catch any kind of `Throwable`, catching an `Error` is rarely appropriate.

Most Exceptions, unlike `Errors`, signify problems you could conceivably recover from. The subclass `RunTimeException` is made up of exceptions that might be thrown by the JVM, such as `NullPointerException`, `ArrayIndexOutOfBoundsException`, and `ClassCastException`.

All exceptions of type `RunTimeExcepotion` or its subclasses are considered to be unchecked Exceptions. And, all exceptions of type Exception other than `RunTimeExcepotion` are checked exceptions. Checked exceptions should be either caught or declared to be thrown.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

/**
 * First experiment using a try/catch block
 */
public class FirstExceptionTest {

   public static void main(String[] args) {
       try{
          // method call may throw exception
          readFile("boogers");

          //this line not executed if exception is thrown
          System.out.println("done");
      } catch (FileNotFoundException e) {
           System.out.println("Executing catch block for: " + e);
      } finally {
           System.out.println("This gets executed no matter what!");
       }
       System.out.println("Main is exiting now");
   }

   public static void readFile(String filename) throws FileNotFoundException{
       Scanner s = new Scanner(new File(filename));
   }

}
```