Exception Handling and Text IO

CSC02A2

Outline

Outline

Exception Handling

Exceptions

Definition

An Exception is an Event that occurs during the execution of a program that disrupts the normal flow of instructions.

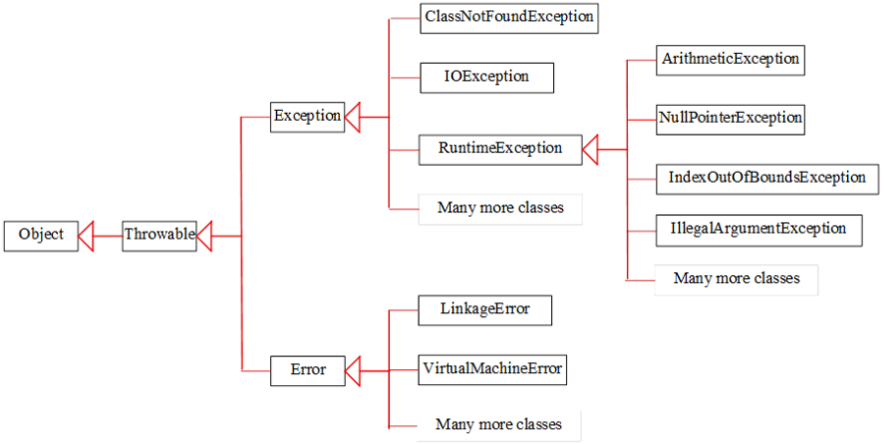
Exception handling is used to enable a programmer to throw an exception to the caller rather than handle the exception locally.

Two types of exceptions exists in Java:

Unchecked exceptions - Exceptions which can occur but the programmer is not forced to handle the exception. Mostly unrecoverable logic errors.

Checked exceptions - Exceptions which can occur and the programmer is forced by the compiler to handle the exception.

Exception types



Handling Exceptions I

Exceptions are handled using a try-catch block:

```
1  public void someMethod()
2  {
3    try
4    {
5      // Statements which can throw exceptions
6    }
7    catch(ExceptionType1 ex)
8    {
9      // Handle exception
10   }
11   catch(ExceptionType2 ex)
12   {
13     // Handle exception
14   }
15   // ...
16 }
```


Handling Exceptions II

Alternatively the exception can be passed along to the caller of the function:

```
1 public void someMethod() throws ExceptionType1, ExceptionType2
2 {
3   // Statements which can throw exceptions
4 }
```

NB

Never use this approach when dealing with exceptions that can occur in the ***main*** method.

Handling Exceptions III

If extra operations need to be performed before an exception is (re)thrown to the caller then the following can be used:

```
1  public void someMethod() throws ExceptionType1
2  {
3    try
4    {
5      // Statements which can throw exceptions
6    }
7    catch(ExceptionType1 ex)
8    {
9      // extra operations
10     throw ex;
11   }
12 }
```


Handling Exceptions IV

If certain operations need to be performed regardless of exceptions being handled in a try-catch block then the finally block is used:

```
1  public void someMethod()
2  {
3    try
4    {
5      // Statements which can throw exceptions
6      // Code that will execute if there are no exceptions
7    }
8    catch(ExceptionType1 ex)
9    {
10     // Code that will execute if there are exceptions
11   }
12   finally
13   {
14      // Code that will always excecute
15   }
16 }
```


When to Use Exceptions

Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.

Be aware, however, that exception handling usually requires more time and resources because it requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods.

Exceptions should only be used when unexpected error conditions occur.

Assertions

An assertion is a Java statement that enables you to assert an assumption about your program. An assertion contains a Boolean expression that should be true during program execution. Assertions can be used to assure program correctness and avoid logic errors.

Assertion should not be used to replace exception handling.

- Exception handling
  - Deals with unusual circumstances during program execution.
  - Addresses robustness of a program.
- Assertions
  - Assure the correctness of the program.
  - Assertions are checked at runtime and can be turned on or off at startup time.

Text I/O

Persistence

Data stored in variables, arrays and objects are stored in volatile primary memory. When program execution stops the data stored is lost. If we require the data to be stored for a long period of time then secondary storage can be used.

Files can store data beyond program execution. Files are usually spilt between textual and binary files. Binary I/O will be discussed in a future lecture.

Files are referred to in two ways:

Absolute filename - Windows path e.g `c:\myUpload\src\Exception.java`

Relative filename - Windows path e.g `src\Exception.java`

The `File` class

The **`File`** class is how Java provides a link to a file which exists on the system.
The **`File`** class is just a *file handle* provided by the operating system.

NB

The **`File`** class cannot read from or write to files.

However, the following methods are available:

- ***`exists()`*** - returns true if the file exists.
- ***`length()`*** - number of bytes used by the file.
- ***`canRead()`*** and ***`canWrite()`*** - Tests permissions for reading and writing.
- ***`isFile()`*** and ***`isDirectory()`*** - Checks if the file handle is actually a file or a directory.
- ***`getRelativePath()`*** and ***`getAbsolutePath()`*** - Get a relative/absolute path to the file.

Writing to text files

The **PrintWriter** class can be used to write text files:

```
1  public void writeToFile() {
2    File textFile = new File("data/myData.txt");
3    // Using PrintWriter with File can throw exceptions
4    PrintWriter txtout = null;
5    try  {
6      txtout = new PrintWriter(textFile);
7      // Write to file like System.out
8      txtout.println("Hello World!");
9    }
10   catch(FileNotFoundException fnfex) {
11     fnfex.printStackTrace();
12   }
13   finally {
14     // Done working with file so close writer
15     if(txtout!=null) txtout.close()
16   }
17 }
```


Reading from text files

The **Scanner** class can be used to read text files:

```
1  public void readFromFile() {
2    File textFile = new File("data/myData.txt");
3    // Using Scanner with File can throw exceptions
4    Scanner txtin = null;
5    try {
6      txtin = new Scanner(textFile);
7      while (txtin.hasNext()) {
8        // Read from file like System.in
9        String line = txtin.nextLine();
10     }
11   }
12   catch (FileNotFoundException ex) {
13     ex.printStackTrace();
14   }
15   finally  {
16     // Done working with file so close writer
17     if (txtin != null) txtin.close();
18   }
19 }
```