File IO in Java

Introduction

Files are used for permanent storage of large amounts of data. There are mainly two types of files:

- **Text file** is file that contains sequence of characters. It is sometimes called ASCII files because the data are encoded using ASCII coding.
- **Binary file** stores data in binary format. The data are stored in the sequence of bytes.

In Java, the standard library which perform I/O (Input/ Output) function is `java.io`. The syntax to import the package is as follow:

```
import java.io.*;
```

According to Package java.io on official Oracle website, `java.io` contains:

- **Class**: `BufferedReader`, `BufferedWriter`, `FilterWriter`, `PrintWriter`, `ObjectInputStream`...
- **Exception**: `EOFException`, `IOException`, `FileNotFoundException`, `SyncFailedException`...

There are plenty others. However, We'll only go over a few. They are enough to finsih the lab. We will discover:

- For **TEXT FILE**
  - `BufferedReader` and `FileReader`
  - `PrintWriter` and `BufferedWriter`
- For **BINARY FILE**
  - `ObjectInputStream` and `FileInputStream`
  - `ObjectOutputStream` and `FileOutputStream`

Text File

If we want to read the file from a `.txt` file, we can use `BufferedReader`.

```
try {
    BufferedReader reader = new BufferedReader(new FileReader("reference.txt"));
} catch (FileNotFoundException ex) {
    ex.printStackTrace();
}
```

BufferedReader

To break down the code, we first look at the **class constructor** of `BufferedReader` on the Class BufferedReader.

```
BufferedReader(Reader in)
BufferedReader(Reader in, int sz)
```

This means that we have to pass a `Reader` class or a class that `extends Reader` to the BufferedReader. We can't simply write `BufferedReader reader = new BufferedReader()`. It will cause `Unresolved Compilation Problem` (compile-time error).

FileReader

`FileReader` is a class that `extends Reader`. So it can be passed to `BufferedReader`'s constructor. It's constructor look like:

```
public FileReader(String fileName) throws FileNotFoundException
public FileReader(File file) throws FileNotFoundException
```

This means that you can either pass a `String` object OR a `File` object.

```
FileReader file_1 = new FileReader("./io_files/reference.txt");
FileReader file_2 = new FileReader(new File("./io_files/reference.txt"));
```

They both function. `File` is another package that provides various powerful tools that allow us to manage the file better. However, to make our course simpler, we'll just use `String` as argument instead of using `File`.

Try-Catch

`FileReader`'s constuctor `throws FileNotFoundException`. This means that the `FileReader` is worried if it is not able to find the file in the given path. So, to handle an exception (or to relieve `FileReder`'s anxiety), we have to follow `handle-or-declare` rule. Since `handle-or-declare` is sophisticated, we won't discuss it here.

> Imagine asking someone to help you find your boyfriend or girlfriend when you don't actually have either of them.

To handle an exception, we use `try-catch` block to embrace the dangerous code that might cause exception.

```
try {
    // dangerous code goes here
} catch (Exception ex) {
    ex.printStackTrace();
}
```

`Exception`

The name of the reference variable (`ex`) need not be `ex`. You might also type `e`, `error`, or `handsome` However, we must employ a meaningful name as part of a proper naming convention. We'll use `ex` throughout the course because it stands for exception (although `e` also signifies the same thing).

```
catch (Exception ex) {} // Compile and it's a good practice
catch (Exception e) {} // Compile and it's a good practice
catch (Exception handsome) {} // Compile but a bad naming practice
```

`.printStackTrace()`

`.printStackTrace()` actually

> Prints this throwable and its backtrace to the standard error stream

Hard to understand? Let's take a look on this example. When I run the following code:

```
public static void main(String[] args) {
    System.out.println(5 / 0);
}
```

It shows:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Tester.main(Tester.java:6)
```

Such error message are known as `Stack Trace`. It shows us:

1. Type of exception
2. Line that cause the exception

This help us debug more efficiently.

Basic Debugging

For example:

- When it throws `java.lang.ArithmeticException: / by zero`, it shows that I have accidentally divide a number by 0. So I can fix it by changing 0 to any integer.
- It shows that the exception happens in `Tester.java` line 6. I can straight going line 6 and fix it.

Finding the line that caused the exception line by line can be challenging in very large projects with hundreds or even thousands of lines. Therefore, by viewing the "StackTrace," we can more readily identify it.

Why using `Try-Catch`?

It lets us to handle the exception with preferable ways. Consider the following code:

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    while(true) {
        System.out.print("Enter two number (-ve value to quit): ");
        int x = scanner.nextInt();
        int y = scanner.nextInt();

        if (x < 0 || y < 0)
            break;

        System.out.printf("%d / %d = %d", x, y, x / y);
    }
    scanner.close();
}
```

if the user enter `5 0`, the exception happens and crashes the program.

```
Enter two number (-ve vaue to quit): 5 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Tester.main(Tester.java:15)
```

However, if we modify the it using `Try-Catch`

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    while(true) {
        try {
            System.out.print("Enter two number (-ve vaue to quit): ");
            int x = scanner.nextInt();
            int y = scanner.nextInt();

            if (x < 0 || y < 0)
                break;

            System.out.printf("%d / %d = %d\n", x, y, x / y);
        } catch (ArithmeticException ex) {
            System.out.println("The second number cannot be 0.");
            continue;
        }
    }
    scanner.close();
}
```

Now, it handle the exception smoothly.

```
Enter two number (-ve vaue to quit): 5 0
The second number cannot be 0.
Enter two number (-ve vaue to quit): 5 5
5 / 5 = 1
Enter two number (-ve vaue to quit): -1 -1
```