# Java Basics Unit

Lesson 9: Simple File I/O

# Lesson 9: Simple File I/O

## Overview

In this lesson, we take a look at how to do simple file input and output. For now, we'll just cover the basics — just enough to get data into and out of files. We'll add more elegant error processing code after we learn about handling exceptions.

## Writing to a File

We are going to use a PrintWriter object to write to our files. There are several other approaches that you can use to write to files but we'll use this one because it is similar to writing output to the Console. The following code declares and initializes a PrintWriter to write to a file called OutFile.txt:

### Notes:

1. If OutFile.txt does not exist, it will be created automatically; and if it does exist, it will be overwritten.
2. We can use println(...) with a PrintWriter in the same way the we use it with System.out.
3. The **flush** method forces everything to be written to the file — PrintWriter may buffer some of our output rather than writing it immediately.
4. The **close** method closes the underlying **stream** that PrintWriter uses to write to the file. It is very important to close resources such as streams — even though we don't have to manually allocate/deallocate memory in Java, we can still have resource leaks if we don't properly clean up after ourselves.

```
PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
out.println("this is a line in my file...");
out.println("a second line in my file...");
out.println("a third line in my file...");
out.flush();
out.close();
```

### Exceptions:

One thing that you'll notice when you type to above code into your program is that NetBeans will highlight an error for the line where the PrintWriter is declared and initialized. Creating a new FileWriter can cause an error called an **IOException**. When we encounter code that can throw an exception (as this code does), we must handle the error in some way. One way to handle this situation is to put a label on our main method that indicates that the code contained in the main method may cause an exception to be thrown. The other option is to write code that will handle the error in some way (perhaps by displaying an error message or by trying to recover from the error).

For now, we are just going to label our code to indicate that it may cause an error. We'll learn how to properly handle exceptions in the next unit. To fix the syntax error, modify the definition of your main method so it looks like this:

```
public static void main(String[] args) throws Exception {
```

## Reading From a File

We are going to use the Scanner object to read from our files. There are alternative approaches to reading from files but we'll use this one because it is similar to reading from the Console. The following code opens the file OutFile.txt, reads each line, and prints each line of the file to the screen:

### Notes:

1. We create the scanner with a BufferedReader instead of System.in (note that BufferedReader requires a new FileReader, so we specify the name of the file we want to read and pass it to the FileReader constructor).
2. Creating a new FileReader can cause an error. Specifically, if the file specified in the constructor does not exist, a FileNotFoundException will be thrown.
3. After the Scanner has been created and initialized properly, we use methods such as nextLine(), just as we do when reading from the Console.

```java
Scanner sc = new Scanner(
    new BufferedReader(new FileReader("OutFile.txt")));
// go through the file line by line
while (sc.hasNextLine()) {
    String currentLine = sc.nextLine();
    System.out.println(currentLine);
}
```

## Wrap-up

That does it for file I/O and the Java Basics Unit. The next unit will cover the object-oriented features of Java including objects, classes, inheritance, interfaces, and more.