File Input

Introduction

In this unit, we are going to learn to read data from files.

Try / Catch / Finally

Try

When coding you are periodically going to use code that is prone to produce an error. When opening or reading data from a file java knows errors may occur. Sometimes the file may not exist or may not have the data we expect it to contain. When your code it likely to cause an error java forces you to put the code in a try block.

A try block will attempt to run the code in it.

- When there is no error, all the code in try block gets run.
- When there is an error, all lines after the error in the try block are skipped and a catch block is executed.

Format:

```
try
{
        // code that may error
}
// catches
```

Catch

Try blocks are required to have at least one catch block. Each catch block can be written to catch a different type of error or you can write a single catch that catches all errors. When using multiple catches the program will run the first catch with a matching error type. A catch statement can be used to correct problems produced by the error and/or print an error message to the user.

Format:

```
catch(ExceptionType name)
{
        // code to fix issues caused by the error
        // code to display an error message
}
```

Note: to print the call stack and the error use the following line of code:

```
exceptionVariable.printStackTrace();
```

Note: A catch with the ExceptionType of Exception will catch all errors.

Single Catch Example:

```
try
{
    int z=y/x;
}
catch(Exception e)
{
    System.out.println("Error you cannot divide by 0");
    e.printStackTrace();
}
```

Multiple Catch Example:

```
try
{
    System.out.println("Enter a whole number: ");
    x = keyboard.nextInt();
    System.out.println("Enter a whole number: ");
    x = keyboard.nextInt();
    int z=y/x;
}
catch(ArithmeticException e)            // catches divide by 0
{
    System.out.println("Divide by 0 Error:");
    e.printStackTrace();
}
catch(InputMismatchException e)         // catches bad input
{
    System.out.println("Input Error:");
    e.printStackTrace();
}
catch(Exception e)                      // catches all other errors
{
    System.out.println("Error:");
    e.printStackTrace();
}
```

Finally

Finally blocks are used to perform any actions that should be run regardless of if there were an error or not. A single block can be placed after the last catch. For example, if you open a file in the try you might want to make sure the file gets close regardless of if there was an error or not.

Format:

```
try
{
        // code to attempt
}
//catches
finally
{
        // code to always run.
}
```

Creating a File Scanner

To read data from a file you must create a Scanner that read from a file instead of System.in. When creating a File Scanner it must be created inside of a try block.

Format for File Scanners:

```
Scanner fromFile = new Scanner(new File("FileName.extention"));
```

Notes:

- **Make sure to put the file in the same location your .class files.**
- **To access file you need the following import**
    - **import java.io.File;**

New Scanner Methods

When working with files we will be using some new Scanner methods to ask the Scanner if there file has more data or not. The new Scanner methods are listed below:

Method	**Description**
hasNextByte()	Returns true if there is a next item and it is of type byte, otherwise it returns false.
hasNextShort()	Returns true if there is a next item and it is of type short, otherwise it returns false.
hasNextInt()	Returns true if there is a next item and it is of type int, otherwise it returns false.
hasNextLong()	Returns true if there is a next item and it is of type long, otherwise it returns false.
hasNextFloat()	Returns true if there is a next item and it is of type float, otherwise it returns false.
hasNextDouble()	Returns true if there is a next item and it is of type double, otherwise it returns false.
hasNextBoolean()	Returns true if there is a next item and it is of type boolean, otherwise it returns false.
hasNext()	Returns true if there is a next item and it is of type String, otherwise it returns false.
hasNextLine()	Returns true if there is a another line of text, otherwise it returns false.

Non-Space Separators:

By default Scanner uses white space to separate items when reading in data, but sometimes items are separate by other things.

To create a File Scanner that separates by non-white space we have to set a delimiter.

Format for setting a delimiter when creating your Scanner:

scannerName = new Scanner(new File(“FileName.extention”)).useDelimiter(“[separators]”);

When listing separators do not separate them. For example, to separate by space, comma and n use: “[,n]”.

Notes:

- **Look at the pattern class to learn more about setting delimiters.**
- **You can also read in a String and t**

Multiple Lines with Non-Space Separators:

Sometimes we will have to read multiple lines of data and each line of data has non-white space separators. To accomplish this feat we will use a File Scanner and feed the lines it scans to a second Scanner.

The follow example is for a program that reads all the numbers in the file and prints their total. The file will have an unknown number of lines with an unknown number of integer values on each line separated by commas.

Example:

```
Scanner fromFile = new Scanner(“Numbers.txt”);

int total = 0;
while(fromFile.hasNextLine())
{
        String s = fromFile.nextLine();
        Scanner fromText = new Scanner(s).useDelimiter("[, ~]");
        while(fromText.hasNextInt())
                total+=fromText.nextInt();
}
System.out.println(total);
```

Terms

Catch	A block of code used to handle errors.
Delimiter	Values used to separate text.
Exception	An Error.
Finally	A block of code that gets run after a try and its catches.
Try	A block of code