Input/ Output

Reading in from the keyboard:

In order to read in data, you must first create a Scanner object. Everything's an object in Java, so scanning in data requires its own object with various methods associated with it that reads in different types of data. The Scanner class is in java.util, so you must inport that at the top of your file.

If you're reading in from the keyboard, then you must create an input stream by reading in from System.in, e.g.,:

Scanner in = new Scanner(System.in);

With this, every character typed in is automatically placed in a buffer, and then the Scanner object in reads data out of that buffer.

The scanner reads in tokens. A token is a string of characters that ends in a whitespace, which includes a blank, a tab, a carriage return (the enter key, or, in a file, a new line), or, in files, an end of file.

The methods available for reading tokens are:

- .nextInt()
- .nextDouble()
- .nextFloat()
- .next() // reads in the next String
- .nextLine() //reads until the end of the line into one long string

Example:

```
Scanner in = new  Scanner(System.in);
System.out.println("Enter an int, a float, a double and a string.");
System.out.println("Separate each with a blank or return.");
int x1 = in.nextInt();
float f1 = in.nextFloat();
double d1 = in.nextDouble();
String s1 = in.nextLine();
System.out.println("Now enter another value.");
String s2 = in.next();
System.out.println("Here is what you entered: ");
System.out.println(x1 + " " + f1 + " " + d1 + " " + s1 + " and " + s2);
in.close();
```

Output:

```
Enter an int, a float, a double and a string.
Separate each with a blank or return.
2  3.1  3.24  hi there how are you
Now enter another value.
no
Here is what you entered:
2 3.1 3.24  hi there how are you and no
```

One more note – after you're finished reading in data, you want to close the input stream. You do this by using the close method associated with the Scanner object. So above, you see the line of code,:

```
In.close();
```

That's closing the input stream, and it's always a good idea to close the input stream when you are done reading in data.

Reading in from Files

In order to read in from a file, you must first create a File object. The File object needs to reference a file that exists on your machine. You can specify the path to the file along with the full file's name, or you can place the file parallel to the src file that holds the java code file you are working on. Once you create a File object, you can use it to read or write from the file, depending on what Type uses it.

If you want to read from a file, you can use the File object you created as input when creating your scanner object instead of "System.in". So, for instance, you'd have:

```
File myFile = new File("myFile.txt");  //myFile.txt is the name of the file on your computer.

if myFile.exists() {
        Scanner in = new Scanner(myFile);
        //…
}
```

myFile.exists is a method that checks to see if the myFile object was successfully created. You're checking to see whether the file exists where you specified it would be, returning True if the file does exist and has been accessed successfully, and False otherwise.

Once you've created a file object, and you've used it as input to your Scanner object, you can then read from the file you've successfully opened. So, just like with reading from the keyboard, you can read data from the file using the same methods, e.g.,:

- int x1 = in.nextInt();
- float f1 = in.nextFloat();
- double d1 = in.nextDouble();
- String s1 = in.nextLine();
- String s2 = in.next();

When you're reading from a file, you may know the general format of the file, but not exactly how long it is. For instance, if you've got a file you're reading in that contains a class roster, it maybe formatted as such:

```
Lastname        firstname        email    class    section   grade
```

But you may not know ahead of time exactly how many lines are in the file, or, you may be reading in many many files with the same format, each with a different length. To take care of the problem of not knowing exactly how long a file is ahead of time, you can use other methods associated with the Scanner object. Some methods that detect whether there is more data in a file are the following:

- .hasNext() // returns true if there is more data to be read
- .hasNextInt() //returns true if the next value is an int

- .hasNextFloat() // returns true if the next value is a float
- .hasNextDouble() // returns true if the next value is a double

You often use the hasNext method to check whether the file has more date.

So, for example, you might have something like this:

```
String name;
int grade;
String email;
File fl = new File(filename);
if (fl.exists()) {
      System.out.println("The file exists!");
      Scanner fn = new Scanner(fl);
      while (fn.hasNext()) {
            name  = fn.next();
            grade = fn.nextInt();
            email = fn.next();
      }
      fn.close();
}
```

Exceptions

An exception is an exceptional event, or something that wasn't supposed to happen when your code runs. So, for instance, dividing by 0, trying to access a file that doesn't exist, reading past the end of an array, or running out of disk space when trying to write to a file are all exceptional events.

In java everything is an object, even the exceptions. When an exception object is created, it contains information about the type of exception (aka error) that occurred.

Some possible exception objects are:

- IOException
- NoSuchElementException
- InputMismatchException
- EOFException
- FileNotFoundException

When we talk about exceptions, we say a program "throws an exception". Thus it makes sense that if a program throws an exception, you should catch the exception and print out information about the exception.

When you're opening and reading from or writing to a file, you will probably want to try executing the code, and, if an error occurs, catch the error and print out an error message. So your code might look like this:

```
File fl = new File(filename);
try {
      Scanner fn = new Scanner(fl);
      int row = 0;
      while (fn.hasNext()) {
            String line = fn.nextLine();
            arr[row] = line;
            row++;
```

```
       }
       fn.close();
 }
catch (FileNotFoundException e) {
    System.out.println(“e.getMessage()");
    System.exit(0);
 }
```

You can catch more than one exception:

```
private int[][] makeMatrix(String filename) {
       int[][] matrix = new int[13][13];
       File fl = new File(filename);
       try {
              Scanner fn = new Scanner(fl);
              int row = 0;
              while (fn.hasNext()) {
                     String line = fn.nextLine();
                     System.out.println(line);
                     String[] numarr = line.split("\t");
                     for (int i = 0; i<numarr.length;i++){
                            matrix[row][i] = Integer.parseInt(numarr[i]);
                     }
                     row++;
              }
              fn.close();
       }
       catch (FileNotFoundException e) {
              System.out.println("File not found: " + e.getMessage());
       }
       catch (IOException e) {
              System.out.println(e.getMessage());
              System.exit(0);
       }
       return matrix;
}
```

If you just want a general exception that catches all exceptions (but doesn’t necessarily give great error messages, you can use the Exception object:

```
try {
       int x = 10;
       int y = 0;
       int z = x/y;
       System.out.println(z);
}
catch (Exception err) {
         System.out.println(err.getMessage());
}
```

As an aside: You can make a particular function throw a particular type of exception.

For example:

```
public void  getFile(int ind) throws FileNotFoundException{
     Scanner in = new Scanner("System.in");
     System.out.println("Enter a filename");
     String s = in.nextInt();
      File f = new File(s);
}
```

The above takes the File not Found exception created when the code tries to create a new file object from the filename string entered and throws it out of the function, so you can catch it in the function that calls this function as opposed to catching it in this function.

Writing to a file:

You've already been writing to the console using System.out.println(). But what if you want to write your data out to a file?

For this, you'll need to make a file object with the name of the file you are writing to, e.g.,:

```
File myOutputFile = new File("myOutputFile.txt");
```

For writing to the file you just created, you'll need to create a PrintStream object with the file object as the input parameter to the constructor, e.g.,

```
PrintStream output = new PrintStream(myOutputFile);
```

Like Scanner objects, PrintStream throws an error, so you'll want to try it and, if something unexpected happens, catch the error and print out the error message. So, for example, your code might look like this:

```
public void makefile() {
      File f1 = new File("testout.txt");
      try {
            PrintStream outfile = new PrintStream(f1);
            for (double[] x: newmatrix) {
                  for (double y: x) {
                        outfile.print(y + " ");
                  }
                  outfile.println();
            }
            outfile.close();
      }
      catch(IOException io) {
            System.out.println(io.getMessage());
      }
}
```

And finally, formatting options

What if you're printing out a file of doubles – maybe average values or values weighted using division? You could end up with a matrix or numbers that looks like this:

```
217.8575927137373 597.655625825856 210.56479046932353 518.4918672220385 980.1225356891215
249.22772143775117 94.27214196883537 218.54436204437766 909.1938505252127 64.77187265528717
522.4332161890065 409.3104756268032 523.8845868006724 545.9574555472581 480.24527204720766

744.5345466483448 480.5374374958481 328.894010297627 6 804.6528435755249 476.5255281033308
701.3804741693555 865.5677865881347 473.16200063038025 939.9005125578615 106.11401024206168
869.7228980147195 107.8829700207542 566.1791646848557 225.14981932646862 555.4267916885535

919.8961924374581 980.4804996320166 930.2115601909552 729.2957731241569 510.5749895400613
661.322074457543 157.8530590786883 155.06158864366725 936.7331953500385 427.63591389063413
357.3626766233403 386.07120201134194 103.29393913692742 138.5895515411516 19.015139669919414

769.2755643999647 451.13010115309584 106.52640929375889 472.5935354330558 118.80427438277653
284.2434781014766 203.21002242875142 287.12133694128306 14.23626895067953 375.5006089897035
259.96665890567016 173.33597287852288 188.71818119248795 644.883318809338 334.60225938402357
```

In some cases, we care about the maximum amount of precision we can get, but there are times when we aren't concerned about that level of precision. We may want to truncate the numbers to keep only the most important values. For this we'd use formatting.

When formatting data that you are going to write out, instead of using of using print or println, you should use format, e.g,

```
System.out.format("");
Outfile.format();
```

Now, when formatting, you specify the total number of digits, and the number of digits after the decimal. So, the following code:

```
Double pi = 3.14159;
System.out.format("%3.2f", pi);
```

Prints:

```
 3.14
```

- %f prints a double or float (out to 6 digits after the decimal)
- %.3f prints a double or float with 3 values after the decimal
- %4.3f prints a double always with 3 values after the decimal (even if it has to add 0s), and then governed by the whole string occupying 4 places.

Now, when printing a number, the digits before the decimal (to the left of the decimal) won't ever be truncated, even if you specify fewer digits when formatting.

However, if the number is smaller than the first number (e.g., %10.3), it will make the whole number 10 characters long, including the decimal, then add spaces to the left.

If you want to add 0s to the left, you could use %010.3f, and that will make the whole printed number 10 digits long, with however many 0s added to the left to make the whole number 10 digits long.

Integer formatting:

To format an integer, you use %d. Again, the integer is never truncated. All digits are considered critical. However, you can add spaces or 0s to the left of the integer through formatting.

- %d prints an integer with as many digits as needed
- %4d prints an integer with as many digits as needed, but always at least 4, with spaces to the left
- %04d prints an integer with as many digits as needed, but always at least 4, with zeros to the left

Examples:

```
System.out.format("%d, %4d, %04d", 34291, 34, 34);
```

Prints:

```
34291,   34, 0034
```

Formatting Strings:

To format strings, you must use %s. Otherwise formatting is similar to integers and doubles:

- %s prints the string (the whole string)
- %15s prints out a string with the specified number of characters, right justified.
- %-15s prints out the string with the specified number, left-justified

E.g.,

```
System.out.format("%s: %15s: %-15s:", "echidna", "wombat", "puffin");
```

Prints:

```
echidna:          wombat: puffin         :
```

Formatting, Special Characters:

Special characters in formatting are preceded with a \

- \t prints a tab
- \n adds a new line
- \\ inserts a backslash
- \’ inserts a single quote
- \” inserts a double quote

E.g.,

```
String[] s = {"cat","dog","echidna","wombat","whale","bunny"};
for (int i = 0; i < s.length; i++) {
	System.out.format("%s\t",s[i]);
	if ((i+1)%2 == 0) {
		System.out.format("\n");
	}
}
```

Prints:

```
      cat             dog
      echidna         wombat
      whale           bunny:
```

E.g., a method that prints out a 3x4 Matrix of integers (random numbers)

```
Random r = new Random();
int arr[][] = new int[3][4];
for (int i = 0; i < arr.length*arr[0].length; i++) {
      arr[Math.floorDiv(i,4)][i%4] = r.nextInt(10);
}
for (int i = 0; i < arr.length; i++) {
      for (int j = 0; j < arr[i].length; j++) {
            System.out.format("%2d\t",arr[i][j]);
      }
      System.out.format("\n");
}
```