File Handling

- Data from input source or output destination is represented in a "stream"
  - A stream is a sequence of binary data (0's and 1's).
- Since bytes are usually the smallest piece of data, Java has Stream objects that read / write data in 8-bit (1 byte) pieces.
- **American Standard Code for Information Interchange (ASCII)**
  - Each character is represented as 1 byte
  - This standard was only keeping English in mind…
- **Unicode**
  - Can represent characters from a larger set
  - We need more than 8 bits to represent characters from different languages.
  - UTF-8 – can store anywhere between 1 – 4 bytes per character
    - Most common standard for the web
  - UTF-16 – can store characters with 2 or 4 bytes
- … and others!

Scanner Object

- In Java, reading from Files and user input from the command line can be done in several different ways.
- **Scanner** is a Java Object that can read values of various types.
- There are several ways to construct / use a Scanner.
- For this class, we will only focus on two cases: Reading user input from the console and parse content within files.
- We've already used Scanner to read user input from the console.
- We can also use Scanners to read data from a file into the application.

```
// If the file does not exist, a FileNotFoundException is thrown
Scanner inFile = new Scanner(new File("file.txt"));
```

- Assume a file called **file.txt** exists in the project root directory containing:

```
This is line 1
This is line 2
This is line 3
```

- Can use an **absolute path**, but this differs based on the File System.
- Can use a **relative path**, which looks within the project directory
- What happens if we pass in the filename String, not a file object?
  - The scanner will use the string (i.e. "file.txt") to parse through!

```
// get all lines in the file and print them one-by-one
while (inFile.hasNextLine()) {
      line = inFile.nextLine();
      System.out.println(line);
}
```

```
// get all words in the file and print them one-by-one
// assuming words are separated by the white space character(s)
while (inFile.hasNext()) {
	word = inFile.next();
	System.out.println(word);
}
```

Tokens

- Tokens are the next "piece" of data you can expect when scanning through the stream.
- .hasNext() checks if the next token is a String
  - but we can directly check certain types
  - .hasNextInt(), .hasNextLong(), .hasNextByte(), .hasNextDouble() …

A note about absolute file paths

- If our file wasn't in our project and in our local hard drive, we can use the file path.
  - Example:
    `/Users/Richert/Desktop/45JWorkspace/Lecture/file.txt`
- For this class, you should always use a relative path, and not the full file path.
- Since the file systems for users of an application vary (not everyone has the same folders / directories / operating systems, …), you can imagine how absolute file paths are not compatible with everyone.

Reading a file using a URL

- If our file was on a webserver somewhere, we can use the URL to get it.
- Example: http://www.ics.uci.edu/~rkwang/file.txt

Example

```
String line;
Scanner remoteIn = null;
try {
	URL remoteFileLocation =
	new URL("http://www.ics.uci.edu/~rkwang/file.txt");

	URLConnection connection = remoteFileLocation.openConnection();
	remoteIn = new Scanner(connection.getInputStream());

	while (remoteIn.hasNextLine()) {
		line = remoteIn.nextLine();
		System.out.println(line);
	}
} catch (IOException e) {
	System.out.println(e.toString());
} finally {
	if (remoteIn != null) {
		remoteIn.close();
	}
}
```

Why would an application want to do this vs. reading something locally?

- If a change in the file needs to be deployed, need to redownload/install entire package if the file is part of the application package.
    - Over The Air (OTA) updates – pulls updated files from remote source(s).
    - No need to redistribute / reinstall existing packages.
- Bottleneck might be network performance or if a server goes down!

Delimiters

- Scanners use delimiters to distinguish what is "next" to read on the input stream.
- By default, Scanners have a delimiter set to any number of whitespace characters.
- If our first line was written as **"This is line 1",** the tokens will be the same.
- Scanner breaks the pieces into non-whitespace tokens.
- We can set Scanner to use a specific delimiter.
    - When inFile.next() is called, it will read the token and put it into a String up to the delimiter or \n character.
    - `inFile.useDelimiter("l");`
    - Example output using "l" as a delimiter:

```
This is
ine 1
This is
ine 2
This is
ine 3
```

- `inFile.useDelimiter("");` is a special case.
    - This is used to parse individual characters.
    - Each inFile.next() call returns a string consisting of the next character.

Example

- Assume we have a fixed format for data in an input file separated by ";".
- [FirstName;LastName;StreetAddress;City;Zip]
- You can obtain "pieces" of data and write general parsing algorithms if there is some fixed format to follow.

// file2.txt

```
Richert;Wang;ICS 424D;Irvine;92697
```

```
String s;
Scanner inFile = null;
try {
      inFile = new Scanner(new File("file2.txt"));
} catch (FileNotFoundException e) {
      System.out.println(e);
}
```

```
String [] stringArray;

// Get the entire line one-at-a-time
while (inFile.hasNextLine()) {
      s = inFile.nextLine();
      stringArray = s.split(";");
      for (int i = 0; i < stringArray.length; i++) {
            System.out.println(stringArray[i]);
      }
}
```

Writing data to a File

- We've talked about reading data from the console and files (remotely and locally). Let's talk about writing data to a file.

```
PrintWriter out = null;
try {
      out = new PrintWriter("output.txt");
      out.println("First Line!");
} catch (FileNotFoundException e) {
      // Non existent path, write access denied, …
      System.out.println(e.toString());
} finally {
      if (out != null) {
            out.close();
      }
}
```

Formatting Data using printf

- System.out.printf() is used to display formatted text.
- System.out.format() behaves the same as System.out.printf().

Example

```
String course = "ICS 45J";
System.out.println("I <3 " + course);

vs.

String course = "ICS 45J";
System.out.printf("I <3 %s", course);
```

'%s' is known as a **format specifier**
s is known as a **conversion character**

%d – decimal integer (or just integer)
%f – floating point
%e – floating point in exponential notation
%c – Characters
%s – Strings
%b – boolean
%% - '%' sign

```
String month = "November";
int day = 9;
int year = 2015;
boolean b = true;
System.out.format("It is %b that today is the %dth of %s in the year
%d.", b, day, month, year);

char grade = 'A';
double percentage = 99.8;
System.out.format("In order to get a %c, I need to score a %f%% on the
final", grade, percentage); // nobody falls in this category ☺
```

Output: In order to get an A, I need to score 99.800000 on the final

- %f has a default precision of 6 decimal places.
- We can define precision using %.[decimal_precision]f
- We can define the precision as follows:

```
System.out.format("In order to get a %c, I need to score %.2f%% on the
final", grade, percentage);
```

Left / Right Justify

- We can define the minimum field width for the amount of space we want a string to consume.
    - Good for aligning output.
- Negative values indicate that the characters should start at the left-most character in the space allotted.
- Positive values indicate that the characters should end at the right-most character in the space allotted.
- If the characters exceed the allotted space, it simply fills in all of the characters (no truncation).
- For any unused characters in the allotted space, a whitespace is inserted.

```
String first = "Richert";
String last = "Wang";
System.out.format("Hello, my name is %-10s %5s", first, last);
```
Output: Hello, my name is Richert   Wang

```
System.out.format("Hello, my name is %-1s %5s", first, last);
```
Output: Hello, my name is Richert _Wang

Writing to a File

- A simple object called PrintWriter exists, which allows you to write content to a file.
- PrintWriter is very simple and doesn't allow you to append to an existing file.

```
PrintWriter out = null;

try {
	out = new PrintWriter("output2.txt");
	String first = "Richert";
	String last = "Wang";
	out.format("Hello, my name is %-10s %5s\n", first, last);
	out.format("Another line");
} catch (Exception e) {
	System.out.println(e.toString());
} finally {
	if (out != null) {
		out.close();
	}
}
```

Append to a File

- Another object, FileWriter, provides more functionality than PrintWriter, including the ability to append to an existing file.

```
FileWriter out = null;
try {
	out = new FileWriter("output", true);
	out.write("First Line!!!");
} catch (FileNotFoundException e) {
	// Non existent path, write access denied, …
	System.out.println(e.toString());
} catch (IOException e) {
	e.printStackTrace();
} finally {
	if (out != null) {
		try {
			out.close();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

ArrayLists

- So far, we've been dealing with Arrays…
  - Arrays need to be defined with a predetermined size.
  - If you try and access / add to a position past the size, an error occurs.
  - Sometimes we don't know what the size limit is and we want to expand our array automatically.
  - ArrayList is the solution to this!

```
// constructs an ArrayList object containing Integer objects
// initially, the list is empty.

ArrayList<Integer> x = new ArrayList<Integer>();
```

- The <Integer> is known as Generic
  - Similar to templates in C++

- Basically, we don't know what an ArrayList can hold and we want to make it general enough where it can hold any type the programmer wants.
    - In this case, we want to hold Integers.
    - But we can also hold Booleans, Doubles, Objects, Scanners, Students, …
    - Without Generics, we would write a generic ArrayList implementation for every possible type?
        - … crazy.

Example: Adding to an ArrayList

```
ArrayList<Integer> x = new ArrayList<Integer>();

x.add(5);
x.add(10);
x.add(15);
x.add(20);

for (int i = 0; i < x.size(); i++) {
      System.out.println(x.get(i));
}
```

Setting items in an ArrayList

- We can change an existing item in the ArrayList using .set()
- `x.set(3,100);`
- Though if we try to set something past the size, we get an error.
    - `x.set(100,100); // ERROR`

Removing from an ArrayList

- We can remove existing elements by index
    - `x.remove(1); // Removes 2nd item in list`
- We can remove existing elements by Object value
    - `x.remove(10); // ERROR? But can't we remove objects?`
    - `x.remove((Integer) 10) // or x.remove(new Integer(10))`
        - In this case, the first occurrence of the **object** is removed from the ArrayList.

Some examples using ArrayLists

```
// Sums up the integers in the ArrayList
public static int sum(ArrayList<Integer> x) {
      int sum = 0;
      for (int i = 0; i < x.size(); i++) {
            sum += x.get(i);
      }
      return sum;
}
```

```
// Swaps two values in the arrayList based on index
public static void swap(ArrayList<Integer> x, int i, int j) {
      int temp = x.get(i);
      x.set(i, x.get(j));
      x.set(j, temp);
}

// Returns the position of the first occurence of the value.
public static int getFirstPosition(ArrayList<Integer> x, int value) {
      int i = 0;
      while (i < x.size()) {
            if (x.get(i) == value)
                  return i;
            else
                  i++;
      }
      return -1;
}
```

ArrayLists containing ArrayLists

- ArrayLists can maintain a list of any Object.
- … including other ArrayLists
- We'll talk more about this kind of structure when talking about 2D arrays.

```
ArrayList<ArrayList<Integer>> listOfLists = new
      ArrayList<ArrayList<Integer>>();

ArrayList<Integer> arrayInt1 = new ArrayList<Integer>();
arrayInt1.add(100);
arrayInt1.add(200);
arrayInt1.add(300);

ArrayList<Integer> arrayInt2 = new ArrayList<Integer>();
arrayInt2.add(-100);
arrayInt2.add(-200);

listOfLists.add(arrayInt1);
listOfLists.add(arrayInt2);

// Print all ints in the listOfLists
for (int i = 0; i < listOfLists.size(); i++) {
      ArrayList<Integer> intList = listOfLists.get(i);
      for (int j = 0; j < intList.size(); j++) {
            System.out.println(intList.get(j));
      }
}
```