# ICS 111
# File Input and Output (I/O)

- Reading from Files
- Writing to Files
- Text input
- Data I/O
- Command-Line Arguments

# Computer Files

- After a program ends its run, all the values in the variables are forgotten
- If desired, values can be stored *persistently* in **files**
  - persistence means the value is remembered beyond the lifetime of the program
  - files can be copied and backed up to provide greater persistence
  - backups are strongly recommended for any important files!
- Files have a **name** and possibly some **data**
- There are different types of data, including **text data** and **binary data**

# Reading a Text File

- A variable of type File represents a file name that we can open for reading or writing

```
java.io.File readFrom = new java.io.File("input.txt");
java.util.Scanner in = new java.util.Scanner(readFrom);
while (in.hasNextLine()) {
  String s = in.nextLine();
  ...
}
in.close();
```

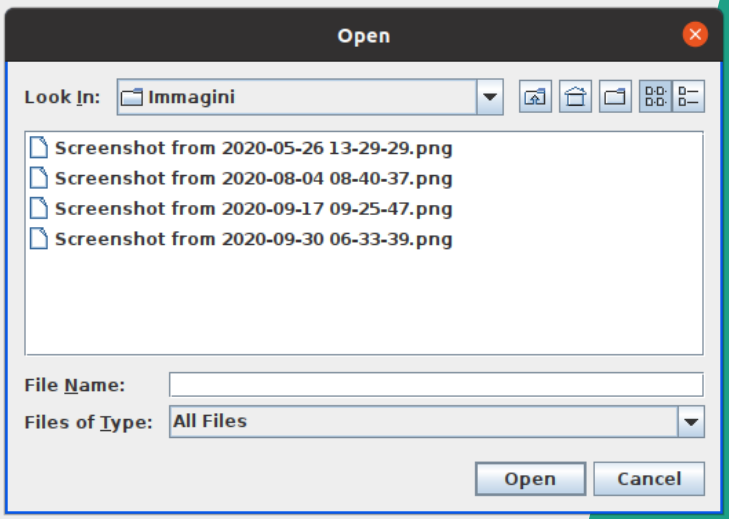
- The constructor for File creates a way for the Scanner to access the file name
- The constructor for Scanner opens the file, which must be closed before the end of the program
  - closing is essential for output files, but not for input files
  - opening (for reading) a file that doesn't exist is an error and results in an exception
- Java doesn't care whether the file name ends with .txt:
  - as long as the code uses Scanner, Java accesses the file as a text file

# File Names, File Chooser

- often we want to give the user a dialog box for choosing a file: `javax.swing.JFileChooser` gives us such dialog boxes
- to ask the user to select a file:

```
JFileChooser c = new JFileChooser();
if (c.showOpenDialog() == JFileChooser.APPROVE_OPTION) {
  File selected = c.getSelectedFile();
  ...
}
```

- For an output file, use `showSaveDialog` instead of showOpenDialog
- After these calls, we do have to check whether the user selected a file
- Try this at home!!!



# Escapes, Backslashes

- If a file name in your code has backslashes, each must be preceded by a second backslash:

```
readFrom = new java.io.File("c:\\hw\\input.txt");
```

- A backslash in Java strings is the **escape** character
  - you are familiar with newlines being written "\n"
  - an escape character gives special meaning to the next character
  - the escape character must itself be escaped when we want it in a string
- Most programming languages have escape characters, allowing us, for example, to include the double quote character inside a string:

```
String answer = "they said \"yes\", all is well";
```

# Writing a Text File

- A writable text file is created (or if it already exists, is emptied) by creating a variable of type `java.io.PrintWriter`:

```
PrintWriter outf = new PrintWriter("output.txt");
```

- Output files must be closed after we are done using them, or we may lose data:

```
outf.close();
```

- In between creating and closing, we can use our usual print functions:

```
outf.println("This line goes into the file");
outf.printf("This line too! counter is %d\n", counter)
```

- In a given program, input files must be separate from output files, otherwise great confusion may ensue

# Binary Data

- text data in a file is a sequence of bytes
- binary data in a file is a sequence of bytes
- in binary data, each byte may have any of the values between 0 and 255, inclusive
  - in text data, bytes may only take the values of printable characters
- binary data bytes may or may not be displayable as printable characters

  !"#*��$ %&#�
- it is OK to read or write a text file with operations for binary data
  - it is not OK to read or write a binary file with text file operations!
  - the results often won't make any sense
- in general, all we want to do with binary data is make copies or compare it for equality
- there may be more specific uses for special kinds of binary data
- especially image and audio files

# Reading and Writing Binary Data

- A java.io.InputStream provides a read operation which returns the next byte
  - the byte is represented as a positive integer 0..255
  - read returns -1 if the read operation has reached the end of the input
  - InputStream has constructors for files and URLs
- There are several types of java.io.OutputStream
- for this class, the interesting one is java.io.FileOutputStream, which has a write operation
  - write takes a byte, represented as an integer in 0..255
- remember to close output streams!

# Constructors for Scanners

- We have seen how to construct a scanner from a file name:

```
Scanner in = new Scanner(new File("input.txt"));
```

- Scanners can also be set up to parse strings:

```
Scanner readString =
  new Scanner("this is the input");
```

- Or the contents of web pages:

```
java.net.URL url = new
java.net.URL("http://hawaii.edu");
Scanner readWebPage = new Scanner(url.openStream());
```

- The scanners work the same no matter what the source

# Java Scanner Methods

- String in.next()  reads the next word (blank-terminated)
- String in.nextLine() reads the next line
- double in.nextDouble() reads the next floating point value
- int in.nextInt() reads the next integer
- every nextX method has a matching boolean hasNextX method that returns whether it is possible to read the corresponding value
  - in.hasNext(), in.hasNextLine(), in.hasNextDouble(), in.hasNextInt()

# Java Delimiters

- String in.next()  reads the next word
- a word is non-blank characters followed by a blank, newline, or the end of input
  - in this case, blank and newline are **delimiters**
- characters that define the beginning or end of a word are known as delimiters
- you can change delimiters for a scanner
- useDelimiter(" yes ") uses the substring " yes " as the delimiter:
  - given the input is "if we say yes I know yes is yes and no is no"
  - in.next() will return the four strings "if we say", "I know", "is", "and no is no".
- in.useDelimiter("") clears the delimiters and tells the scanner's `next` method to return strings that are a single character long
  - containing the next character in the input

# Regular Expressions

- useDelimiter can be told to use groups of characters as delimiters
- [ square brackets ] identify groups of characters
- `in.useDelimiter("[0-9]");` uses any digit as the delimiter
- `in.useDelimiter("[.,;:]");` tells the scanner that in.next() should return all the input up to the next one of these punctuation marks
- `in.useDelimiter("[^a-zA-Z0-9]");` means to use as a delimiter any non-alphanumeric character
  - the initial ^ indicates a negation, so “use as delimiter any character that does not belong to the character ranges in the brackets”
- the argument to useDelimiter is a regular expression
  - regular expressions are a general way of capturing patterns in strings
  - regular expressions are more general than discussed here
  - regular expressions are used outside of Java in shell programming and in string matching
  - regular expressions are of interest in the theory of programming languages: part of the syntax of language definitiions is usually expressed as regular expressions

# Character Classes

- Several Java methods tell us whether a character is a digit, a letter, upper or lower case, etc

  Character.isWhiteSpace(char c)

  Character.isDigit(char c)

  Character.isLetter(char c)

  Character.isUpperCase(char c)

  Character.isLowerCase(char c)
- In each case, these methods return a boolean that is true if the character belongs to that group, and false otherwise
- Many more can be found at https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Character.html

- String.trim removes any initial or final blanks:
- String withBlanks = "   Hello world  ";
- String withoutBlanks = withBlanks.trim();  // withoutBlans is "Hello World"

# Parsing Numbers

- We can parse a string to an integer or a double
- However, the number must fill the entire string (except for any initial or terminating blanks):
- Double.parseDouble("3.1415") is fine
- Double.parseDouble(" 3.14+3") is not
- If using a scanner, can test with hasNextInt() or hasNextDouble() before calling nextInt() or nextDouble()

# printf formats

- printf prints according to a format string
- % in the format string indicates a value taken from one of the later arguments to printf:
  - %s: print a string
  - %d: print a decimal integer
  - %f: print a floating point number such as 3.1
  - %e: print a floating point number with the exponent, such as 3.1e+0
  - %g: print a floating point number with the best of the preceding two notations
  - %x: print an integer in hexadecimal
  - %%: print a % sign (there is no later argument corresponding to %%)

```
printf (“my name is %s: %d + %f is %f%%\n”, myName, 2, 3.0, 2.3);
```

# printf format width and alignment

- Between the % and the format character may be a number, which specifies the format width (in characters)
- %3d, print an integer with one or two leading blanks if necessary
  - examples: `"1234", " 12", "  1"`
  - numbers that don't fit in the format width are still printed in their entirety
- %-3d, print an integer with one or two following blanks if necessary
  - examples: `"1234", "12 ", "1  "`
- %03d, print an integer with one or two leading 0s if necessary
  - examples: `"1234", "012", "001"`
- %5.2f, print a floating point number using (at least) 5 characters, with exactly two digits after the decimal point
  - examples: `" 3.14", "139.00"`
- %(5d, print negative numbers in (parentheses)

# Command-Line Arguments

- We have seen that the command-line arguments are given to main in its array of strings parameter
- When an argument begins with a "-" character, it is usually an option or a flag
  - e.g. "-v" or "--verbose" to tell the program to print more debugging information
- If argument order doesn't matter, we can process the command-line arguments with an enhanced for loop:

```
for (String a: args) { ...
```

- Arguments are often file names

# File Names in Command-Line Arguments

Arguments are often file names. Here is a simple program that just prints the contents of all files named in its arguments:

```
public static void main(String[] args) {
  for (String a: args) {
    printFileContents(a);
  }
}

public static void printFileContents(String fileName) {
  java.io.File f = new java.io.File(fileName);
  java.util.Scanner in = new java.util.Scanner(f);
  while (in.hasNextLine()) {
    System.out.println(in.nextLine());
  }
}
```

- do this at home: try to run this program before going on to the next slide

# Exceptions and `throws`

- The code on the preceding slide won't compile, because creating a scanner from a file may cause an exception called FileNotFoundException
- An exception is said to be **thrown**
  - later we will see how to **catch** exceptions!
- For now, we can keep the compiler happy by simply declaring the exceptions that each method may throw

```
public static void main(String[] args) throws java.io.FileNotFoundException {
  for (String a: args) {
    printFileContents(a);
  }
}

public static void printFileContents(String fileName) throws java.io.FileNotFoundException {
  java.io.File f = new java.io.File(fileName);
  java.util.Scanner in = new java.util.Scanner(f);
  while (in.hasNextLine()) {
    System.out.println(in.nextLine());
  }
}
```

# Substitution Cipher

- A simple way to encrypt is to just choose a letter a fixed distance away from the letter we are encrypting
- Caesar cipher: A -> D, B -> E, ... Z -> C
  - “hello world” becomes “khoor zruog”
  - decryption uses the same substitution table, backwards
  - this is an easy cipher to break, so it is no longer seriously used
- since English has 26 letters, we can swap the two halves of the alphabet: A -> M, B -> N, ... Z -> L
  - this is “rot13”, where the letters are rotated through the alphabet by 13 positions
  - then decryption is the same operation as encryption

# Summary

- This lecture expands on previous knowledge about text input and output
- Reading from files and writing to files is intentionally very similar to reading from the user and printing to the display
- Scanners and parseInt/parseDouble provide many ways of using user and file input, and printf provides much flexibility for output
- It is easy to read and output files!
- Try it at home: read a web page from a web server
  - maybe http://www2.hawaii.edu/~esb/2020fall.ics111/oct12-transcript.txt
  - output it to the screen, and also save it to a file.