Files
Modern computers have many different kinds of storage components:
- memory (aka RAM or DRAM)
- disks (aka “hard disks” or “hard drives”)
- caches
- external media (USB sticks, DVD, CD-RW, etc.)

Why so many?
Because they behave differently!
## Memory vs. Disk

<table>
<thead>
<tr>
<th>Typical Properties</th>
<th>Memory</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity:</td>
<td>1-4 Gb</td>
<td>&gt;100 Gb</td>
</tr>
<tr>
<td>When power goes off:</td>
<td>Data is lost</td>
<td>Data is safe</td>
</tr>
<tr>
<td>When program ends:</td>
<td>Data is lost</td>
<td>Data is safe</td>
</tr>
<tr>
<td>Sequential access speed:</td>
<td>1.2 GB / second</td>
<td>50 MB / second</td>
</tr>
<tr>
<td>Random access speed:</td>
<td>1.2 GB / second</td>
<td>??</td>
</tr>
<tr>
<td></td>
<td></td>
<td>~ 66.7 seeks / second</td>
</tr>
</tbody>
</table>

**Tradeoff:**

Disks have greater capacity (more GB) and offer permanent storage; Memory is much faster.
Files and Directories

- Your operating system (Windows, Mac, Linux, etc.) divides your disk hierarchically.
- This creates a tree structure.
  - Each element in the tree is a *folder* or *directory*.
  - Inside directories are *files*.

```
C:\
  - Documents and Settings
    - tmp
    - super3
    - All Users
  - Program Files
    - Adobe...
    - WinShell
```

Hello.java
Files and Variables

- Recall variables:
  - They have types, names, and a location in memory
  - You can put data inside them, and use the data later

- Files are similar abstractions, but for the disk:
  - They have names and a location on the disk
  - You can put (lots of) data inside them
  - You can later retrieve that data and use it
Two main file operations

- **Read**
  - Move data from a file on the disk into a variable (or variables) in memory

- **Write**
  - Move data from a variable (or variables) in memory to a file on the disk

- Two tricky details to these operations –
  - Data types
  - Checked Exceptions
**File class**

- The `File` class is a template for objects (in memory) that represent actual files (on disk):
  
  ```java
  File myFile = new File("C:\myfile.txt");
  ```

- The `File` class is in the `java.io` package. To use it, include the import declaration:
  
  ```java
  import java.io.*;
  ```

- `io` (or I/O) stands for input/output.
Creating a Scanner for a file, general syntax:

```java
Scanner <name> = new Scanner(new File("<file name>"));
```

Example:

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

Instead of getting data from the keyboard via `System.in`, this Scanner object gets data from the file `numbers.txt` in the current folder (directory).
The following program does not compile:

```java
import java.io.*; // for File
import java.util.*; // for Scanner

public class ReadFile {
    public static void main(String[] args) {
        Scanner input = new Scanner(new File("data.txt"));
        // do something
    }
}
```

The compiler reports:

```
ReadFile.java:6: unreported exception
java.io.FileNotFoundException; must be caught or declared to be thrown
```
Exceptions

- **exception**: An object representing a program error.
  - Programs with invalid logic will cause ("throw") exceptions.

- Examples:
  - Trying to read a file that does not exist.
  - Dividing by 0.
  - Using `charAt(10)` on a string of length 5.
Checked exceptions

- **checked exception**: An exception that must be explicitly handled (otherwise the program will not compile).
  - We must either:
    - handle ("catch") the exception, or
    - explicitly state that we choose not to handle the exception (and accept that the program will crash if the exception occurs)
throws clause: How to waive your rights

- **throws clause**: Tells the compiler that a method may throw an exception.
  - Like a waiver of liability:
    "I hereby agree that this method might throw an exception, and I accept the consequences (crashing) if this happens."

- **throws clause, general syntax**:
  ```java
  public static <type> <name>(<params>) throws <type> {
  ```

- **Example**:
  ```java
  public static void main(String[] args)
  throws FileNotFoundException {
  ```
import java.io.*;     // for File, FileNotFoundException
import java.util.*;   // for Scanner

public class ReadFile {
    public static void main(String[] args) throwsFileNotFoundException {
        Scanner input = new Scanner(new File("data.txt");
        // do something
    }
}
Recap: Tokens

- The Scanner breaks apart the input into tokens. It will interpret the tokens in different ways depending on if you call `next()`, `nextInt()`, or `nextDouble()`.

- Assuming the following input file:

  23   3.14
  "John Smith"

  The tokens in the input can be interpreted as the given types:

<table>
<thead>
<tr>
<th>Token</th>
<th>Type(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. 23</td>
<td>int, double, String</td>
</tr>
<tr>
<td>2. 3.14</td>
<td>double, String</td>
</tr>
<tr>
<td>3. &quot;John&quot;</td>
<td>String</td>
</tr>
<tr>
<td>4. Smith&quot;</td>
<td>String</td>
</tr>
</tbody>
</table>
The input cursor

- Consider a file that contains this text:
  
  308.2
  14.9  7.4  2.8
  3.9  4.7  -15.4
  2.8

- A Scanner views all input as a stream of characters, which it processes with its input cursor:
  
  308.2\n  14.9  7.4  2.8\n  3.9  4.7  -15.4\n  2.8\n  ^
Consuming tokens

- Each call to `nextInt`, `nextIntInt`, `nextDouble`, etc. advances the cursor to the end of the current token, skipping over any whitespace. Each call *consumes* the input.

```java
input.nextDouble();
308.2\n 14.9 7.4 2.8\n\n\n3.9 4.7 -15.4\n2.8\n```

```java
input.nextDouble();
308.2\n 14.9 7.4 2.8\n\n\n3.9 4.7 -15.4\n2.8\n```
Consider an input file named `input.txt`:

```
308.2
14.9 7.4  2.8
3.9 4.7 -15.4
2.8
```

Write a program that reads the first 5 values from this file and prints them along with their sum.

**Output:**

```
number = 308.2
number = 14.9
number = 7.4
number = 2.8
number = 2.8
Sum = 337.1999999999999
```
// Displays the first 5 numbers in the given file, 
// and displays their sum at the end.

import java.io.*;  // for File, FileNotFoundException
import java.util.*; // for Scanner

public class Echo {
    public static void main(String[] args)
        throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers.dat"));
        double sum = 0.0;
        for (int i = 1; i <= 5; i++) {
            double next = input.nextDouble();
            System.out.println("number = " + next);
            sum += next;
        }
        System.out.println("Sum = " + sum);
    }
}
Reading a whole file

- The preceding program is assumes you know how many values you want to read.

- How could we read in ALL of the numbers in the file, without knowing beforehand how many the file contains?
The `Scanner` has useful methods for testing to see what the next input token will be.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasNext()</td>
<td>whether any more tokens remain</td>
</tr>
<tr>
<td>hasNextDouble()</td>
<td>whether the next token can be interpreted as type <code>double</code></td>
</tr>
<tr>
<td>hasNextInt()</td>
<td>whether the next token can be interpreted as type <code>int</code></td>
</tr>
<tr>
<td>hasNextLine()</td>
<td>whether any more lines remain</td>
</tr>
</tbody>
</table>
Exercise: Version 2

- Rewrite the previous program so that it reads the entire file.

**Output:**

```plaintext
number = 308.2
number = 14.9
number = 7.4
number = 2.8
number = 3.9
number = 4.7
number = -15.4
number = 2.8
Sum = 329.29999999999995
```
// Displays each number in the given file,  
// and displays their sum at the end.

import java.io.*;   // for File, FileNotFoundException
import java.util.*; // for Scanner

public class Echo2 { 
    public static void main(String[] args) 
        throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers.dat"));
        double sum = 0.0;
        while (input.hasNextDouble()) {
            double next = input.nextDouble();
            System.out.println("number = " + next);
            sum += next;
        }
        System.out.println("Sum = " + sum);
    }
}
Exercise: Version 3

- Modify the preceding program again so that it will handle files that contain non-numeric tokens.
  - The program should skip any such tokens.

- For example, the program should produce the same output as before when given this input file:

  308.2  hello
  14.9 7.4  bad stuff 2.8

  3.9 4.7  oops  -15.4
  :-)    2.8  @#*($&
Solution: Version 3

// Displays each number in the given file,  
// and displays their sum at the end.

import java.io.*;   // for File, FileNotFoundException
import java.util.*; // for Scanner

public class Echo3 {  
    public static void main(String[] args)   
        throws FileNotFoundException {  
            Scanner input = new Scanner(new File("numbers.dat"));  
            double sum = 0.0;  
            while (input.hasNext()) {  
                if (input.hasNextDouble()) {  
                    double next = input.nextDouble();  
                    System.out.println("number = " + next);  
                    sum += next;  
                } else {  
                    input.next();   // consume / throw away bad token  
                }  
            }  
            System.out.println("Sum = " + sum);  
    }  
}
Exercise

- Write a program that accepts an input file containing integers representing daily high temperatures.

Example input file:
42 45 37 49 38 50 46 48 48 30 45 42 45 40 48

- Your program should print the difference between each adjacent pair of temperatures, such as the following:
  - Temperature changed by 3 deg F
  - Temperature changed by -8 deg F
  - Temperature changed by 12 deg F
  - Temperature changed by -11 deg F
  - Temperature changed by 12 deg F
  - Temperature changed by -4 deg F
  - Temperature changed by 2 deg F
  - Temperature changed by 0 deg F
  - Temperature changed by -18 deg F
  - Temperature changed by 15 deg F
  - Temperature changed by -3 deg F
  - Temperature changed by 3 deg F
  - Temperature changed by -5 deg F
  - Temperature changed by 8 deg F
import java.io.*;
import java.util.*;

public class Temperatures {
    public static void main(String[] args)
        throws FileNotFoundException {
        Scanner input = new Scanner(new File("weather.dat"));
        int temp1 = input.nextInt();

        while (input.hasNextInt()) {
            int temp2 = input.nextInt();
            System.out.println("Temperature changed by "+
                (temp2 - temp1) + " deg F");
            temp1 = temp2;
        }
    }
}
Line-based processing
Who's next in line?

- Reading a file line-by-line, general syntax:

```java
Scanner input = new Scanner(new File("<file name>"));
while (input.hasNextLine()) {
    String line = input.nextLine();
    <process this line>;
}
```

- The `nextLine` method returns the characters from the input cursor's current position to the nearest \n character.
Reading between the newlines

23 3.14 John Smith "Hello world"
   45.2  19

23\t3.14 John Smith\t"Hello world"
\t\t45.2  19\n
- input.nextLine()
23\t3.14 John Smith\t"Hello world"
\t\t45.2  19\n   

- input.nextLine()
23\t3.14 John Smith\t"Hello world"
\t\t45.2  19\n   

- NB: The \n character is consumed but not returned.
Exercise

- Write a program that reads a text file and "quotes" it by putting a > in front of each line.

**Input:**

Hey Prof. Yates,

I would like to know more about files. Please explain them to me.

Sincerely,

Susie Q. Student

**Output:**

> Hey Prof. Yates,

> I would like to know more about files. Please explain them to me.

> Sincerely,

> Susie Q. Student
import java.io.*;
import java.util.*;

public class QuoteMessage {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("message.txt"));
        while (input.hasNextLine()) {
            String line = input.nextLine();
            System.out.println(">"+ line);
        }
    }
}


Example

Example file contents:

123 Susan 12.5 8.1 7.6 3.2
456 Brad 4.0 11.6 6.5 2.7 12
789 Jennifer 8.0 8.0 8.0 8.0 7.5

Consider the task of computing the total hours worked for each person represented in the above file.

Susan (ID#123) worked 31.4 hours (7.85 hours/day)
Brad (ID#456) worked 36.8 hours (7.36 hours/day)
Jennifer (ID#789) worked 39.5 hours (7.9 hours/day)
Line-based or token-based?

- Neither line-based nor token-based processing works.

- The better solution is a hybrid approach
  - Break the input into lines.
  - Break each line into tokens.
A Scanner can be constructed to tokenize a particular String (such as one line of an input file).

Scanner <name> = new Scanner(<String>);

Example:
String text = "1.4 3.2 hello 9 27.5";
Scanner scan = new Scanner(text);  // five tokens
Scanner input = new Scanner(new File("<file name>"));
while (input.hasNextLine()) {
    String line = input.nextLine();
    Scanner lineScan = new Scanner(line);
    <process this line>;
}
Multi-line records

- The following data represents students' course information.

  Erica Kane
  3 2.8 4 3.9 3 3.1
  Greenlee Smythe
  3 3.9 3 4.0 4 3.9
  Ryan Laveree, Jr.
  2 4.0 3 3.6 4 3.8 1 2.8

  Each student's record has the following format:
  - **Name**
  - **Credits  Grade  Credits  Grade  Credits  Grade  ...**

- How can we process one or all of these records?
File output
Outputting text to files

- **PrintWriter**: A class in the `java.io` package that lets you print output to a destination such as a file.

- Setting up an output file, general syntax:
  ```java
  PrintWriter <name> =
  new PrintWriter(new File("<file name>"));
  ```

- Example:
  ```java
  PrintWriter output = new PrintWriter(new File("output.txt"));
  output.println("Hello, file!");
  output.println("This is a second line of output.");
  ```
Your program should close what it has opened!
  Use the `close()` method to close a file after your done with it.

Example:
```java
PrintWriter out = new PrintWriter("output.txt");
out.println("Hello, output file!");
out.close();
```

Also works on `Scanner` objects
`close()` releases system resources associated with an open file.
Exercise

- Write a method named `Copier` that takes two filenames and copies the contents from the first file into the second file.
public class Copier {
    public static void main(String [] args)
        throws FileNotFoundException {
        Scanner keyboard = new Scanner(System.in);
        String name1 = keyboard.next();
        String name2 = keyboard.next();

        Scanner input = new Scanner(new File(name1));
        PrintWriter output = new PrintWriter(new File(name2));

        while (input.hasNextLine()) {
            output.println(input.nextLine());
        }

        input.close();
        output.close();
    }
}
Binary File I/O
Not all files are text files

- Images, videos, mp3s, oh my!
- Let’s say we wanted to write a program that makes a copy of an mp3 –

Q: What’s wrong with using `Scanner` and `PrintWriter`?

A: If the file contains no `\n` characters, the `nextLine()` method will try to read the entire file into memory all at once!
Reading binary data from files

- **FileInputStream**: a class in the `java.io` package that lets you read binary data in chunks whose size you specify.

- Setting up an input file, general syntax:
  ```java
  FileInputStream <name> =
  new FileInputStream(new File("<file name>"));
  ```

- Example:
  ```java
  FileInputStream input = new FileInputStream(new File("input.mp3"));
  int nextByte = input.read();
  byte [] buffer = new byte[100];
  int numBytesRead = input.read(buffer);
  ```
Outputting binary data to files

- **PrintStream**: Another class in the `java.io` package that lets you print output to a destination such as a file.
- **Writer**: text output, **Stream**: binary output
- **System.out** is a PrintStream object!
  - Any methods you have used on `System.out` (such as `print`, `println`) will work on every PrintStream object.
Setting up the PrintStream

- Setting up an output file, general syntax:
  
  ```java
  PrintStream <name> =
  new PrintStream(new File("<file name>"));
  ```

- Example:
  
  ```java
  PrintStream output = new PrintStream(new File("output.txt"));
  output.println("Hello, file!");
  output.println("This is a second line of output.");
  
  Or
  
  output.write(buffer, offset, numBytes);
  ```
PrintStream warning

- **Caution**: Do not open a file for reading (`Scanner`) and writing (`PrintStream`) at the same time.
  - You could overwrite your input file by accident!
Exercise

- Write a class named `BinaryCopier` that reads in two filenames from the user and copies the *binary* contents from the first file into the second file.
public class BinaryCopier {
    public static void main(String[] args) throws FileNotFoundException, IOException {
        Scanner kb = new Scanner(System.in);
        String name1 = kb.next();
        String name2 = kb.next();
        FileInputStream input =
            new FileInputStream(new File(name1));
        PrintStream output =
            new PrintStream(new File(name2));

        byte[] buffer = new byte[100];
        int numBytesRead = input.read(buffer);

        while (numBytesRead > -1) {
            output.write(buffer, 0, numBytesRead);
        }
        input.close();
        output.close();
    }
}