



Week 04: File systems, File I/O, and Exceptions

CS 220: Software Design II — D. Mathias

Why Files?

Files allows us to store data “permanently” on a computer

what happens to the values of program variables when your program shuts down?

what about the values of files when your computer shuts down?

Files and directories (i.e., folders) form the basis of computer organization

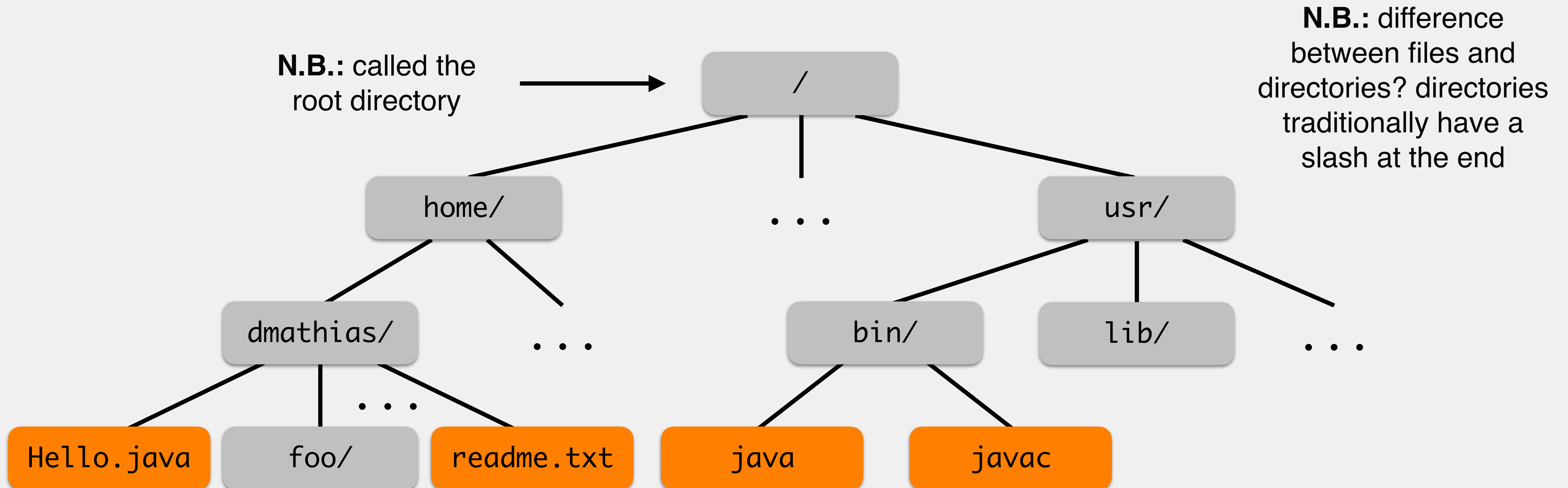
files go in directories

directories can go inside other directories

The File System

File systems dictate how to organize files and directories (and drives)

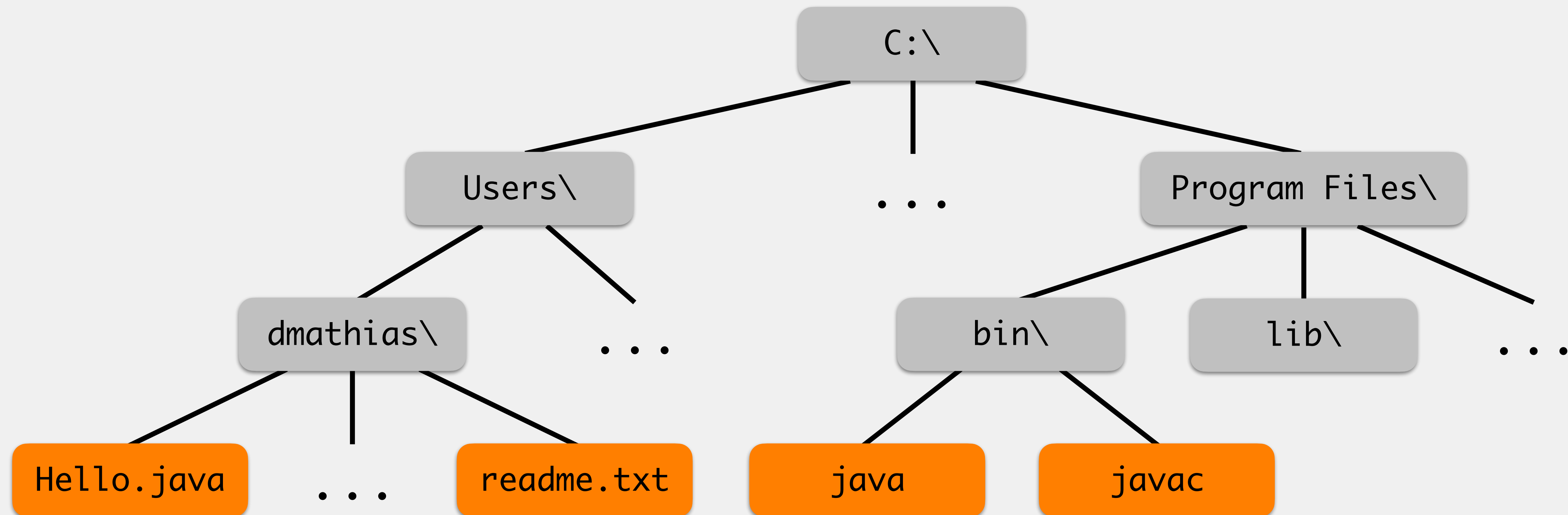
Modern computers are comprised of a hierarchical file system



The File System (Windows)

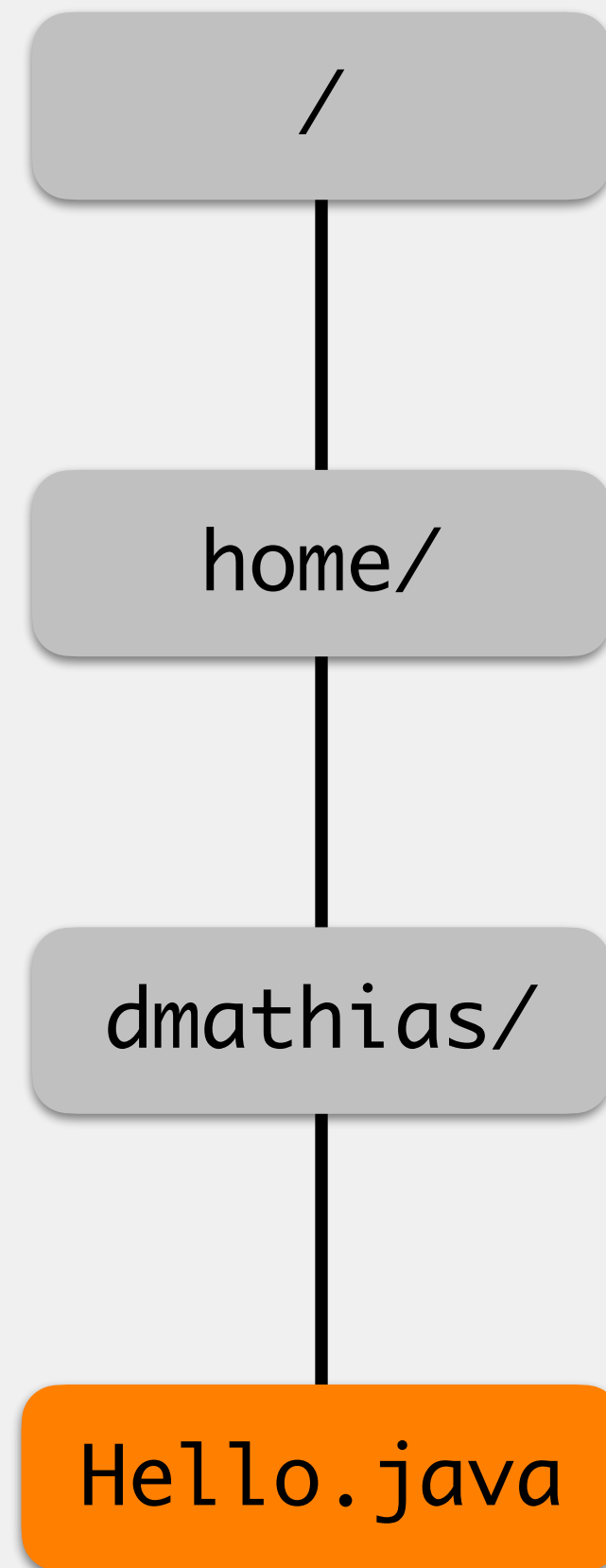
Windows people, you will need to use backslashes

and you'll need to escape them in strings, e.g., "C:\\Users\\dmathias\\Hello.java"



Absolute Paths

N.B.: very easy to look this path up on your machine!



Need some way of referring to a particular location in the system

absolute path: the full address of the file/folder

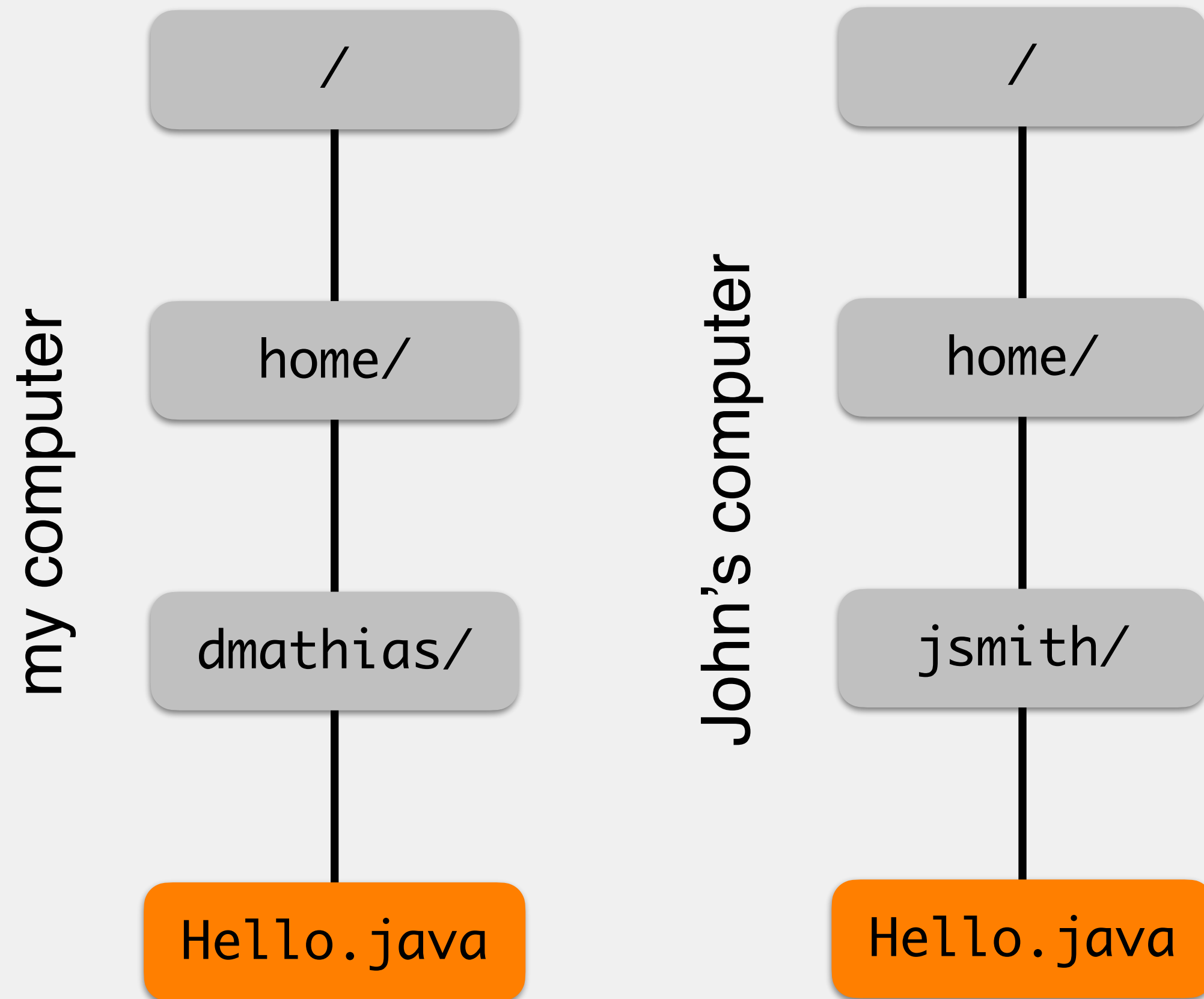
concatenates every point between the root and the desired location

e.g., `/home/dmathias/Hello.java`

e.g., `/home/dmathias/`

e.g., `C:\Users\dmathias\Hello.java`

Problems with Absolute Paths



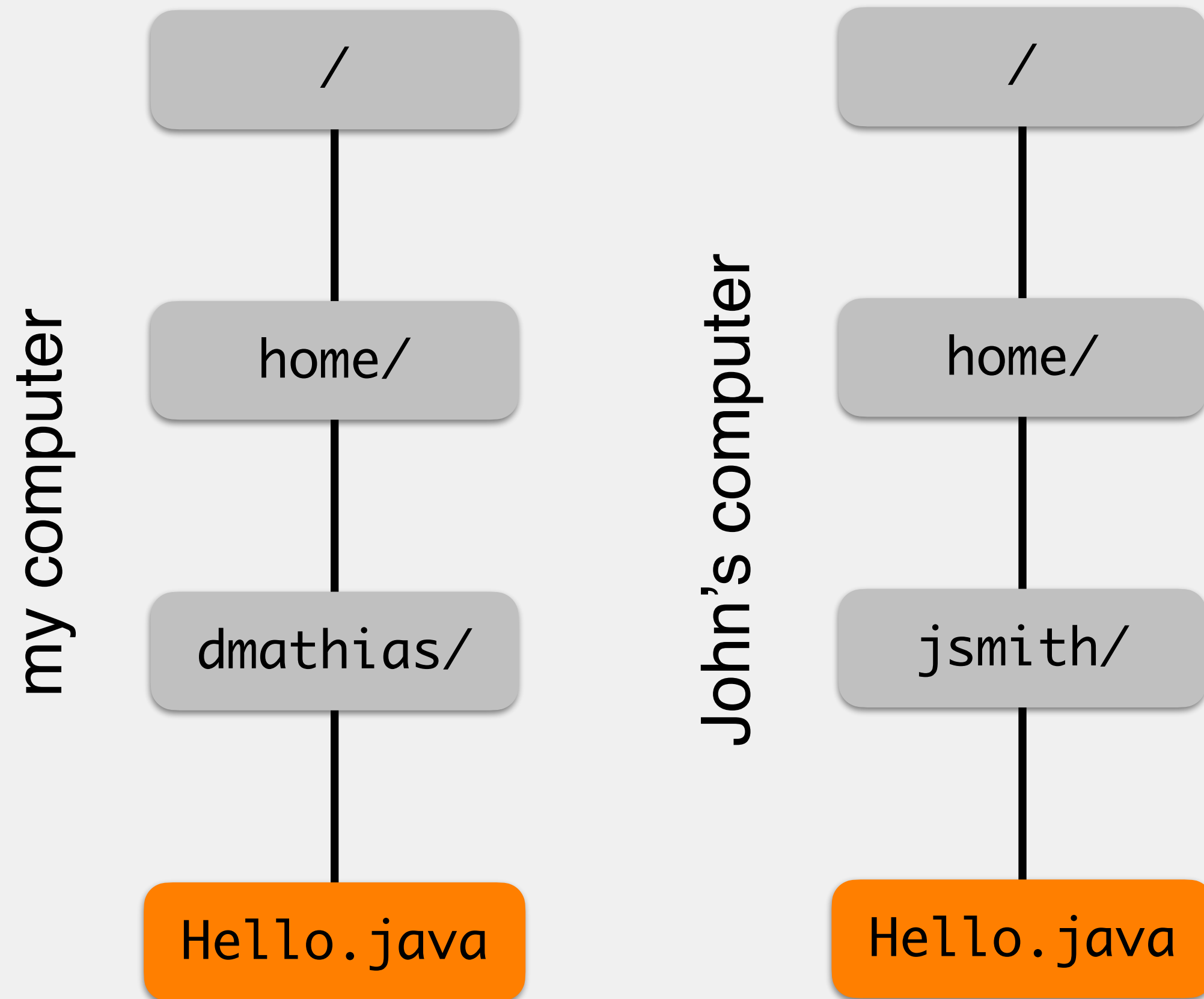
Referring to a file in absolute terms will likely not work across computers

`/home/dmathias/Hello.java` doesn't exist on John's computer

cannot run code referring to that file on John's computer

Need some way to refer to files relative to the computer they are on

Relative Paths



relative path: refers to the location of a file/directory relative to where your program is currently working

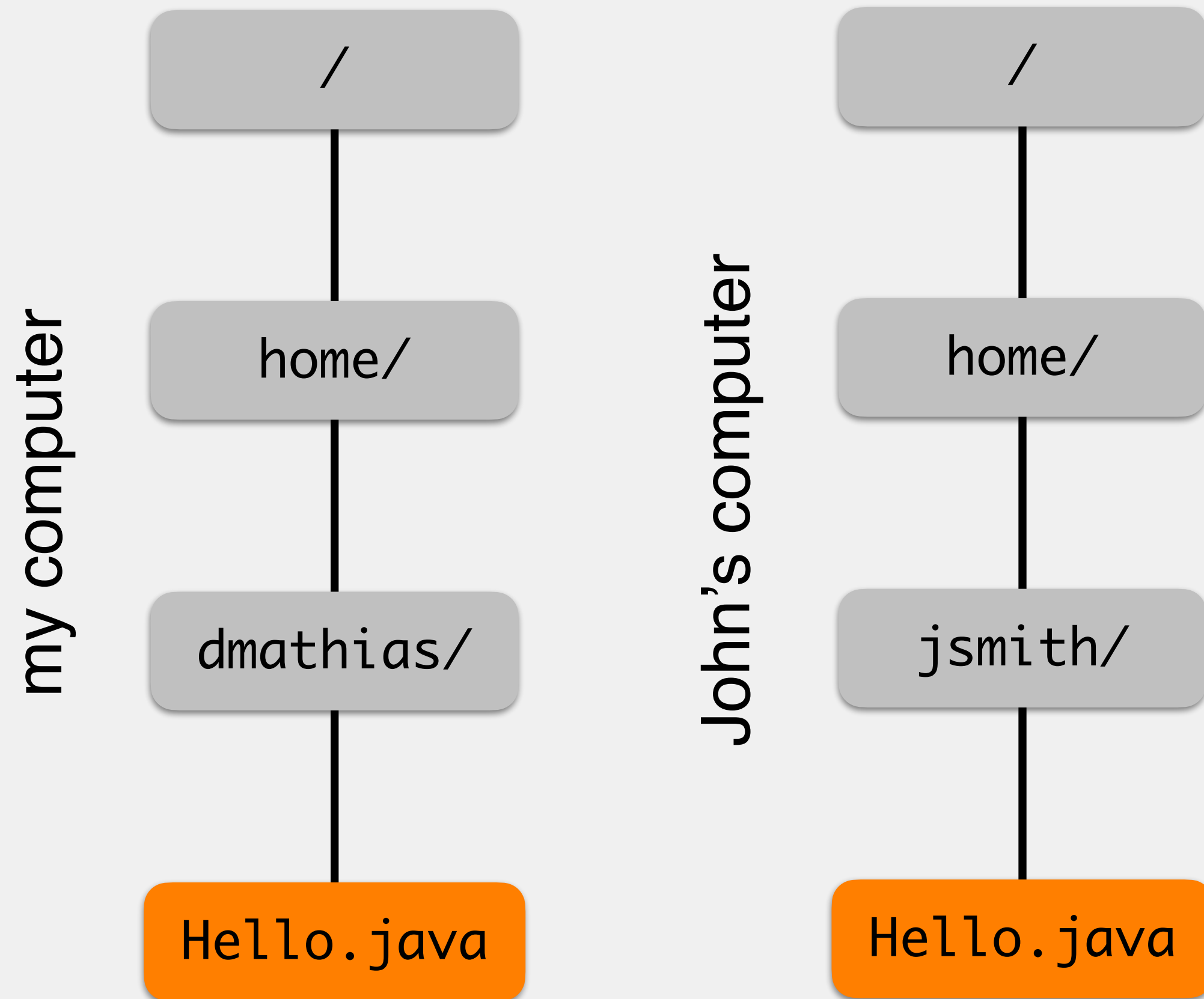
working directory: the current path of where a program is run

e.g., Hello.java's working directory is dmathias and jsmith respectively

./ is shorthand for the current working directory

./Hello.java

Relative Path Shorthand



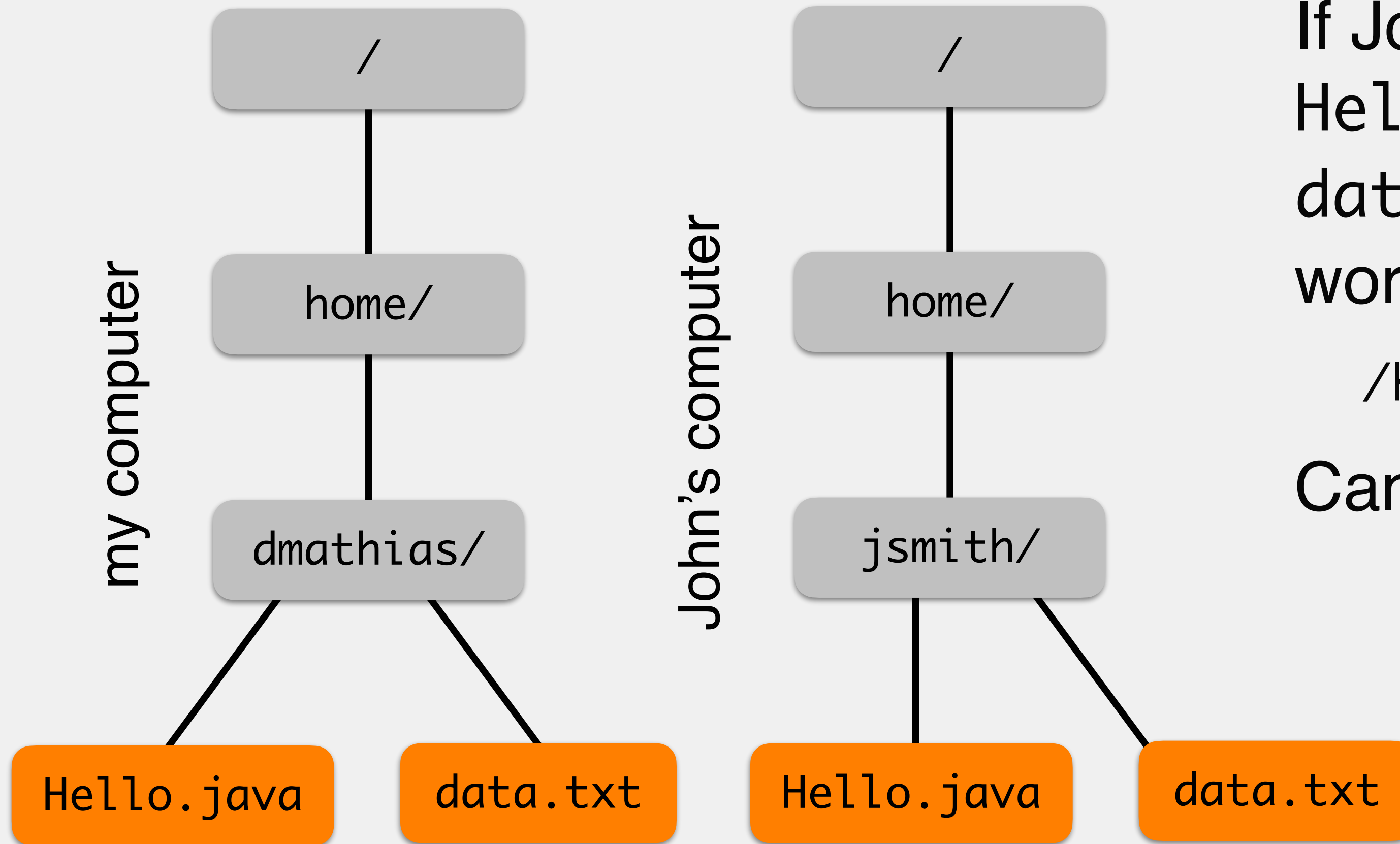
When we run `Hello.java`, the path `./Hello.java` would be interpreted as...

`/home/dmathias/Hello.java` on my computer

and as

`/home/jsmith/Hello.java` on John's computer

Relative Paths



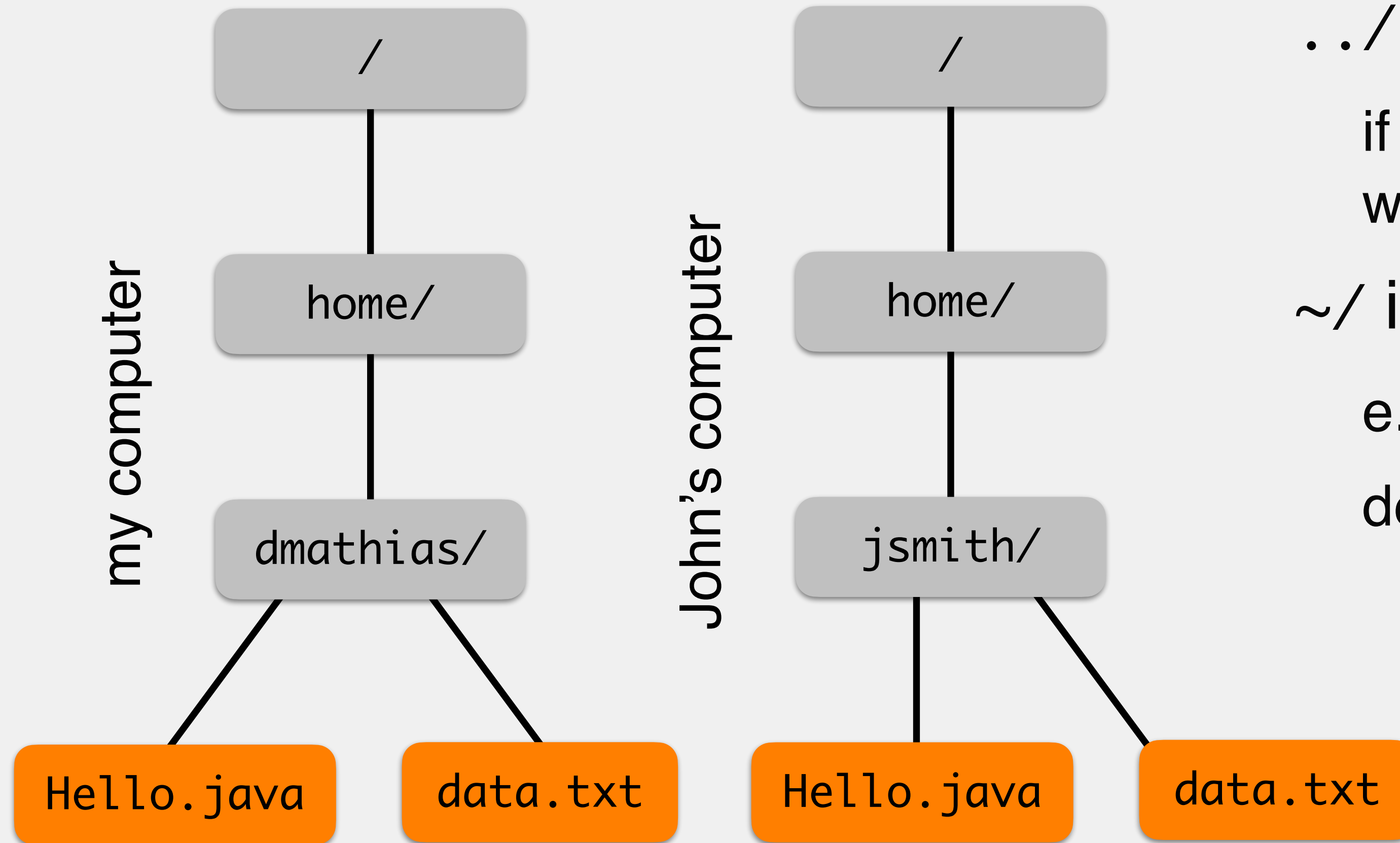
If John is running my copy of `Hello.java` and we want to access `data.txt`, absolute paths will not work

`/home/dmathias/data.txt` does not exist

Can use relative paths!

`./data.txt` will work on either computer

Additional Shorthand for Relative Paths



`../` is the working directory's parent
if we were running `Hello.java`, `../..`
would refer to the root

`~/` is the home directory

e.g., `dmathias`, `jsmith`
defined by computer

Should not need these if you
arrange your files correctly

You see them more in CS270

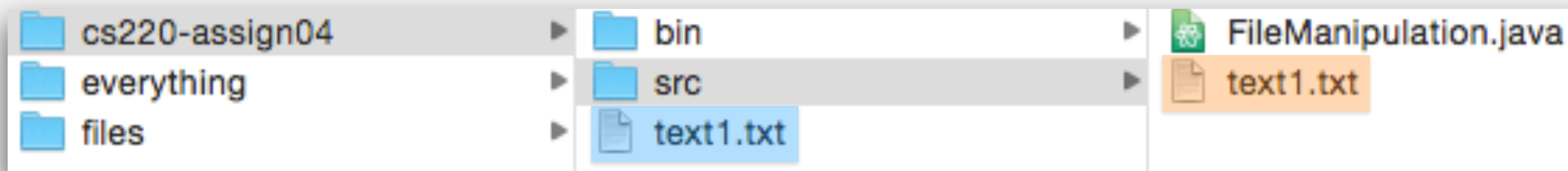
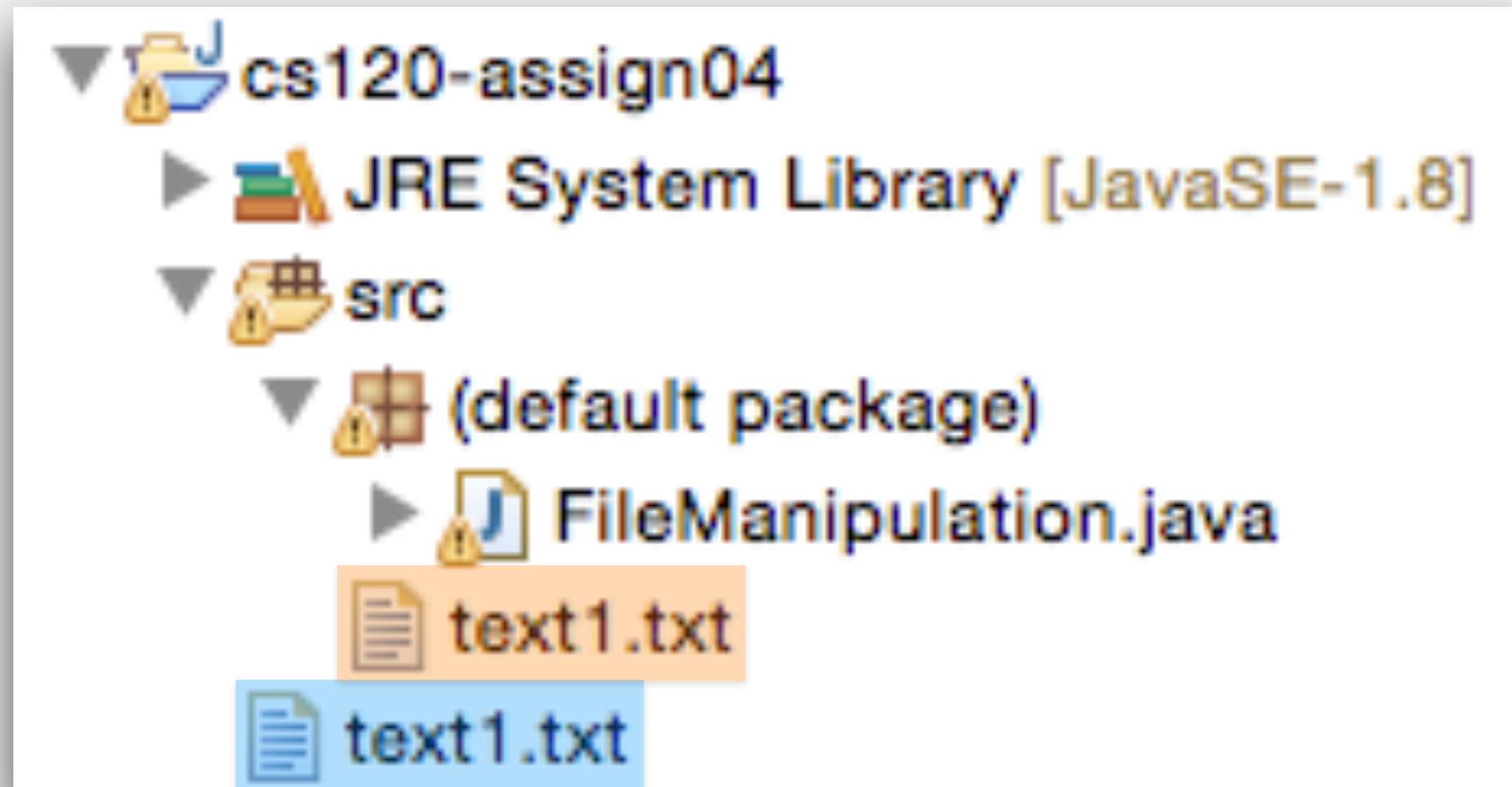
Arranging Files in Eclipse

Working directory isn't always clear

Eclipse's working directory is your project, **not** where the Java file is

The blue file is in the working directory, the tan-is file is not

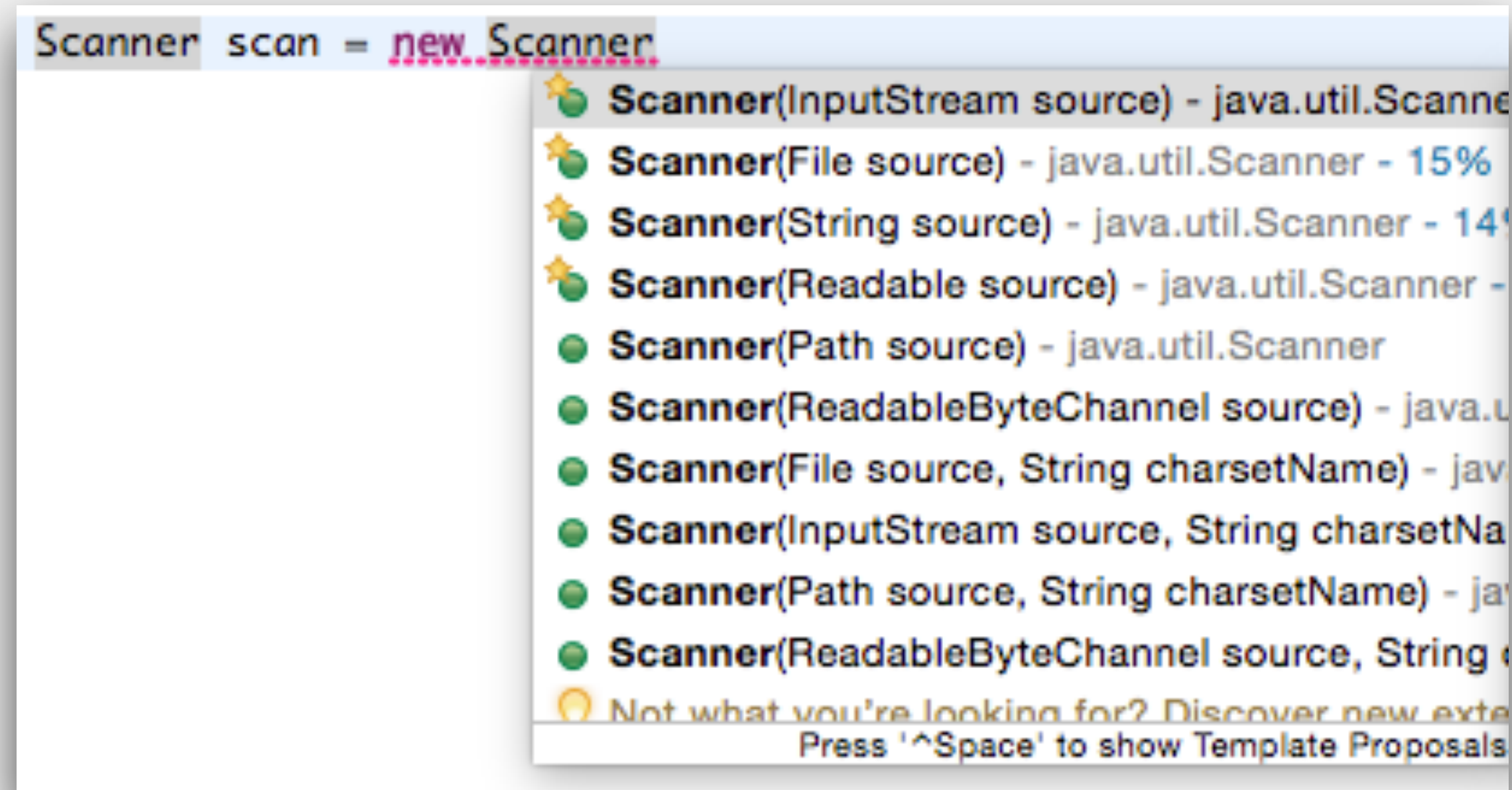
Can access files elsewhere, just tricky to construct the path



File I/O

- Reading and writing files is a core need for modern programs
 - aka *file input and output*, or *file i/o*
- Modern programming languages provide constructs for handling this
 - Java has options

The Scanner Class



N.B.: you can even pass in a String to the constructor and use your usual Scanner methods to parse the String

Provides input from the console... and many other sources

Input source defined by the constructor

System.in is an InputStream

File allows us to access a system file

Excellent example of method overloading!

The File Class

parameter: the (absolute or relative) path of the file to read (String)

```
File dataFile = new File(<path>);
```

File objects have many methods associated with them (e.g., `isDirectory()`, `isFile()`), but we will only need the object itself in CS 220

Using Scanner for Console Input vs Files

```
Scanner scan = new Scanner(System.in);
String firstName;

System.out.print("What is your first name? ");
firstName = scan.nextLine();
System.out.print("Your name is ");
System.out.println(firstName);
```

```
What is your first name? Jim
Your name is Jim
```

name.txt

Jim

```
Scanner scan = new Scanner(new File("name.txt"));
String firstName;

firstName = scan.nextLine();
System.out.print("Your name is ");
System.out.println(firstName);
```

```
Your name is Jim
```

Reading Files with Scanner

- Can use all of the Scanner methods you're familiar with in files
 - e.g., `next()`, `nextLine()`, `nextInt()`
- But how can you tell when you're done reading?
 - a file can have an arbitrary number of lines

name.txt

```
Jim  
Patty  
Claire  
Rob  
Josh  
Jessie
```


Detecting the End of File (EOF) with Scanner

Various Scanner methods return a boolean depending on whether or not there is another value

```
scan.hasNextLine();    //returns true if there is another String line
```

```
scan.hasNextInt();     //returns true if there another int
```

```
scan.hasNext();        //returns true if there is another String
```

```
Scanner scan = new Scanner(new File("names.txt")); // instantiates File also
while(scan.hasNextLine()) {
    String line = scan.nextLine();
    System.out.print(line);
}
```

Detecting the End of File (EOF) with Scanner

Why instantiate the File object within the Scanner constructor call

```
Scanner scan = new Scanner(new File("names.txt")); // instantiates File also
while(scan.hasNextLine()) {
    String line = scan.nextLine();
    System.out.print(line);
}
```

Notice that this is the only place we will use that File object. Therefore, there is no need to give it a name (create a variable) for it.

What if the file doesn't exist?

Important note on file existence

A filename includes the complete path to the file:

data.txt is **not** the full filename

/home/users/dmathias/cs220/programs/program01/data.txt is the name

It is sometimes confusing to see an error message that says a file doesn't exist when you can see the file in your directory. **Remember, it's not saying that no file with that "partial file name" (such as data.txt) exists, it's saying that none exists with the full path name you've specified.**

Errors & Exceptions

Errors are catastrophic problems that cannot be recovered from
e.g., missing semicolon, not declaring variables, improper method overloading
detected before the program is run

Exceptions are problems we **might** be able to recover from
e.g., division by zero, manipulating null values, accessing nonexistent files
detected during execution of the program

Two Categories of Exceptions

Checked exceptions are used for **predictable** but **unpreventable** problems that are **reasonable** to recover from

e.g., a file is corrupted and cannot be read, a database is not accessible

failure is outside of programmer's control, but might be fixable

e.g., asking for a different file, retrying connection to the database

Unchecked exceptions are everything else, usually problems of a logical nature

i.e., programming failure

e.g., divide by zero, index out of bounds

we have mostly seen unchecked exceptions thus far (some saw checked in CS120)

Two Categories of Exceptions

- The “checking” for checked and unchecked exceptions refers to **compile-time**.
- Whether an exception is checked or not is determined by Java.
- When writing code that may result in a checked exception, **the programmer must handle that exception in some way**.
 1. use the throws keyword to pass-the-buck (so to speak)
 2. use a try/catch block to deal with the exception more directly
- Failure to do one of these things will result in compilation errors.**

Handling Exceptions

Thus far, exceptions cause the program to crash

not an unreasonable reaction for unchecked exceptions

Java provides a construct to prevent a program crash by **trying** some piece of code and **catching** an exception if it occurs

optionally, we might **finally** do something, regardless of whether an exception was caught

allows program to continue running if we want

Java requires use of this construct if a checked exception is possible

try/catch Block

Allows programmers to catch and handle exceptions

required for checked exceptions

optional for unchecked exceptions

```
Scanner scan = new Scanner(System.in);
boolean repeat;
int value;
do {
    repeat = false;
    System.out.print("Enter a number: ");
    try {
        > value = scan.nextInt();
    } catch (InputMismatchException e) {
        System.out.print("Not a number. ");
        System.out.println("Try again.");
        > repeat = true;
    }
} while(repeat);
>
```

```
Enter a number: hello
Not a number. Try again.
Enter a number: 42
```

try/catch Block

```
try {  
    Scanner scan = new Scanner(new File("data.txt"));  
    value = scan.nextInt();  
} catch(FileNotFoundException e) {  
    // do something  
} catch(InputMismatchException e) {  
    // do something  
} catch(NoSuchElementException e) {  
    // do something  
} catch(IllegalStateException e) {  
    // do something  
}
```

N.B.: can have multiple catch statements; will evaluate in order (like if/else if) and will only execute the first one found that is true

All file i/o is checked

what if the file isn't there?

what if we can't open it?

what if we try to access beyond the end of the file?

what if our Scanner is closed?

try/catch/finally Block

```
Scanner scan = new Scanner(System.in);
int value = 0;
System.out.print("Enter a positive number: ");

try {
    value = scan.nextInt();
} catch(InputMismatchException e) {
    System.out.println("Not a positive number.");
} finally {
    scan.close();
}

System.out.println("Your number: " + value);
```

finally will always execute

- if something is caught
- if nothing is caught

try/catch/finally Block

```
Scanner scan = new Scanner(System.in);
int value = 0;
System.out.print("Enter a positive number: ");

try {
    value = scan.nextInt();
} catch (InputMismatchException e) {
    e.printStackTrace();
} finally {
    scan.close();
}

System.out.println("Your number: " + value);
```

If we want to catch, then terminate the program and print the stack trace, we can

default behavior of Eclipse if you ask it to construct a try/catch block

try/catch/finally Block

```
Scanner scan = new Scanner(System.in);
int value = 0;
System.out.print("Enter a positive number: ");

try {
    value = scan.nextInt();
} catch(InputMismatchException e) {
    throw new InputMismatchException();
} finally {
    scan.close();
}

System.out.println("Your number: " + value);
```

Can also rethrow the exception

A catch statement *resolves* an exception that exists

Rethrowing the exception means that there is a *new* exception to resolve

Creating Our Own Exceptions

Exception is a class, just like other Java classes

We already know several descendants

`ArrayIndexOutOfBoundsException`, `IOException`, `InputMismatchException`

All extend `Exception` in some way

unchecked exceptions extend `RuntimeException`, which extends `Exception`

Extending Exception

```
public class SwearingException extends Exception {  
    public SwearingException() {  
        super("No swearing!");  
    }  
}
```

```
public void printText(String text) throws SwearingException {  
    Pattern pattern = Pattern.compile("[@$*!#%&]{3,}");  
    Matcher m = pattern.matcher(text);  
    if (m.find()) {  
        throw new SwearingException();  
    } else {  
        System.out.println(text);  
    }  
}
```

Must override constructor

just pass a message to super

can also override constructor that uses a message parameter

Throwing the custom exception only requires instantiating the new object and throwing it

Methods throwing an exception must modify their signature

Extending Exception

```
public class SwearingException extends Exception {  
    public SwearingException() {  
        super("No swearing!");  
    }  
}
```

```
public void printText(String text) throws SwearingException {  
    Pattern pattern = Pattern.compile("[@$*!#%&]{3,}");  
    Matcher m = pattern.matcher(text);  
    if (m.find()) {  
        throw new SwearingException();  
    } else {  
        System.out.println(text);  
    }  
}
```

Must override constructor

just pass a message to super

can also override constructor that uses a message parameter

Throwing the custom exception only requires instantiating the new object and throwing it

Methods throwing an exception must modify their signature

Scanner Details

- Pros

- can parse/cast text right away (e.g., `next()`, `nextLine()`, `nextInt()`)
- simple

- Cons

- cannot handle binary files
- very small buffer
 - can only read a relatively small number of chars at a time
- cannot be used in a multi-threaded program
- can be a pain in the neck when handling input with different types (e.g. Strings and ints)



WELL, PRINCE, so Genoa and Lucca are now just family estates of the Buonapartes. But I warn you, if you don't tell me that this means war, if you still try to defend the infamies and horrors perpetrated by that Antichrist I really believe he is Antichrist I will have nothing more to do with you and you are no longer my friend, no longer my 'faithful slave,' as you call yourself! But how do you do? I see I have frightened you sit down and tell me all the news."

It was in July, 1805, and the speaker was the well-known Anna Pdvlovna Scherer, maid of honor and favorite of the Empress Marya Fedorovna. With these words she greeted Prince Vasili Kurdgin, a man of high rank and importance, who was the first to arrive at her reception. Anna Pdvlovna had had a cough for some days. She was, as she said, suffering from lagrippe; grippe being then a new word in St. Petersburg, used only by the elite.

All her invitations without exception, written in French, and delivered by a scarlet-liveried footman that morning, ran as follows:

"If you have nothing better to do, Count [or

All data is stored in binary (0s and 1s)

Data must be *encoded* for its format

determines the translation from what we see to 0s and 1s, and back again

The encoding applied to data can be determined by the file extension

e.g., .png for pictures, .txt for text files

tells computer what program can interpret that encoding

Data Encoding: An Analogy

```
String text = "42";
```



```
00110100 00110010
```

Both store the same data (to us), but
in different formats

Format dictates how the computer:
interprets the data
manipulates the data

```
int num = 42;
```



```
00101010
```

N.B.: simplified
binary to more easily
see the difference

Text vs Binary Files

Text files store all data as chars – even numbers!

- we'll use the .txt extension in this class

- advantage: text files are simpler for humans to work with

Binary files store all data according to its type

- can only store primitive types

- we'll use the .bin extension in this class

- advantage: binary files are more space-efficient, easier for computers to work with

Both file types are ultimately rendered as 1s and 0s

- how we interpret the 1s and 0s depends on the file type

Primitive Data Types in Java

Integer Numeric Types (can only be whole numbers)

| | | | | |
|-------|---------|----------------------|---------|---------------------|
| byte | 1 byte | -128 | through | 127 |
| short | 2 bytes | -32678 | through | 32677 |
| int | 4 bytes | -2147483648 | through | 2147483647 |
| long | 8 bytes | -9223372036854775808 | through | 9223372036854775807 |

Decimal Numeric Types (can be whole or decimal numbers)

| | | |
|--------|---------|-------------------------------|
| float | 4 bytes | 7 decimal digits of accuracy |
| double | 8 bytes | 15 decimal digits of accuracy |

Character Type

| | | |
|------|---------|------------------------|
| char | 2 bytes | any keyboard character |
|------|---------|------------------------|

Logical Type

| | | |
|---------|--------|---------------|
| boolean | 1 byte | true or false |
|---------|--------|---------------|

An int Stored in a Text vs Binary File

1503478923
^^^

number.txt

number.bin

10 chars x 2 bytes/char = 20 bytes

An int Stored in a Text vs Binary File

1503478923



number.txt

10 chars x 2 bytes/char = 20 bytes

number.bin

1 int x 4 bytes/int = 4 bytes

Many ints Stored in a Text vs Binary File

1 billion 10 digit ints stored in a file

numbers.txt

10 chars x 2 bytes/char x 1 billion
= 20 billion bytes
= 20 GB

numbers.bin

1 int x 4 bytes/int x 1 billion
= 4 billion bytes
= 4 GB

Text vs Binary Files

Choose a text file if...

- data is mostly text (some numbers are OK)

- easy human access is important

 - i.e., anyone can just open and read the file

- data isn't too large

Choose a binary file if...

- data is mostly numbers

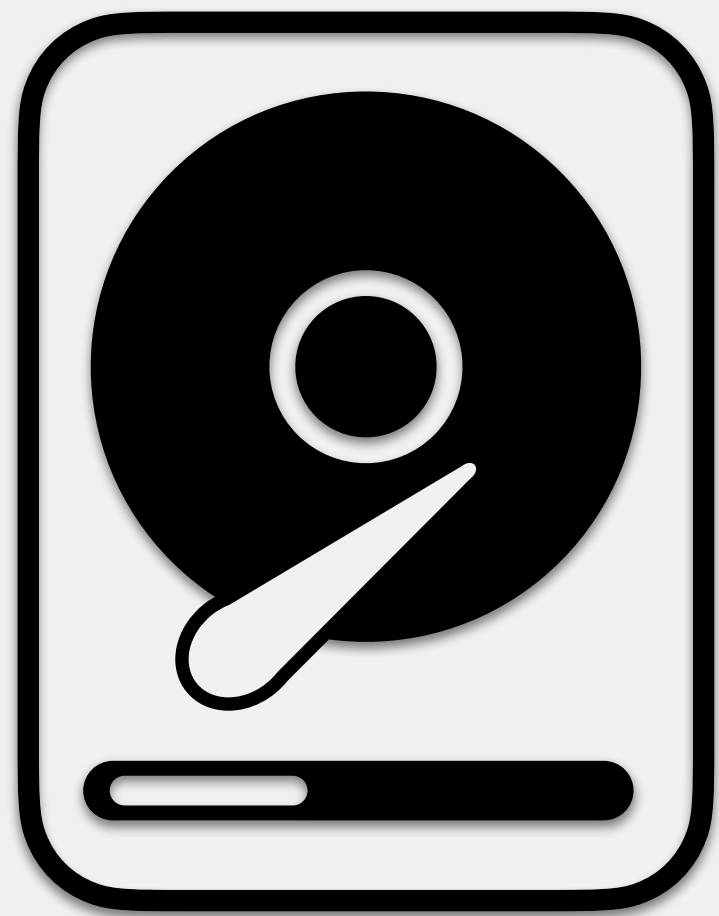
- data won't be read by humans/will be interpreted by another program

- large amounts of data

BufferedReader/FileReader

Java classes used together to read **text** files

cannot read binary files!



FileReader

reads data
from file



BufferedReader

stores data in buffer
for when program needs it

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.Scanner;

public class FileManipulation {

    public static void main(String[] args) {
    }

    /**
     * Copies the .txt files from the dir
     */
    private static void copyFiles() {

        File srcDir = new File("files/");
        File destDir = new File("./");

        String files[] = srcDir.list();

        try {
            for (String file : files) {
                File sFile = new File(src, file);
                File dFile = new File(dest, file);
            }
        }
    }
}
```

File I/O Exceptions

`FileNotFoundException`: indicates file was not found

produced any time we try to create a new `File`

`IOException`: indicates some i/o problem occurred

several exceptions extend from this — including `FileNotFoundException`

Both are checked exceptions

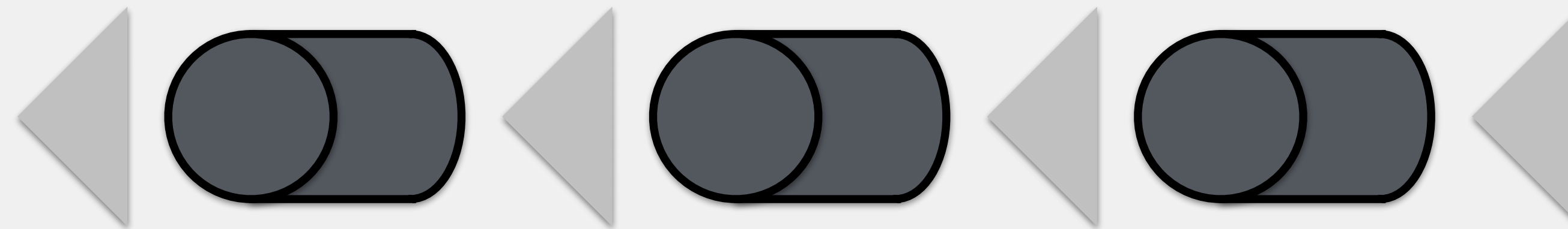
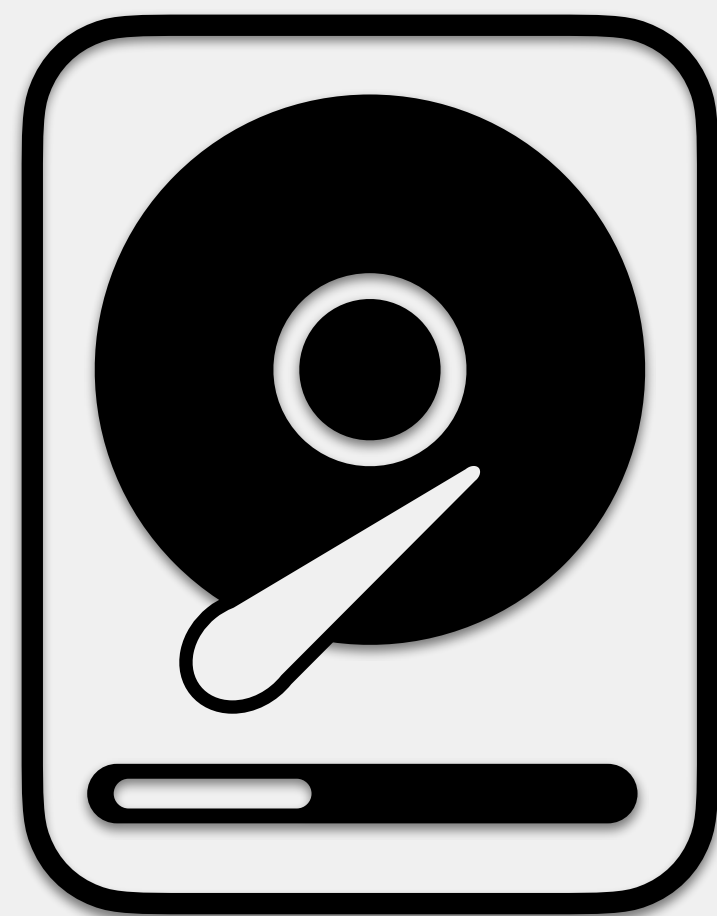
Order of catch matters

catching `IOException` first will mean `FileNotFoundException` will also be caught there

BufferedWriter/FileWriter/PrintWriter

Java classes used together to write to **text** files

cannot write to binary files!



FileWriter

writes out text

BufferedWriter

buffers text
for output

PrintWriter

access to print
and println

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.Scanner;

public class FileManipulation {

    public static void main(String[] args) {
    }

    /**
     * Copies the .txt files from the dir
     */
    private static void copyFiles() {

        File srcDir = new File("files/");
        File destDir = new File("./");

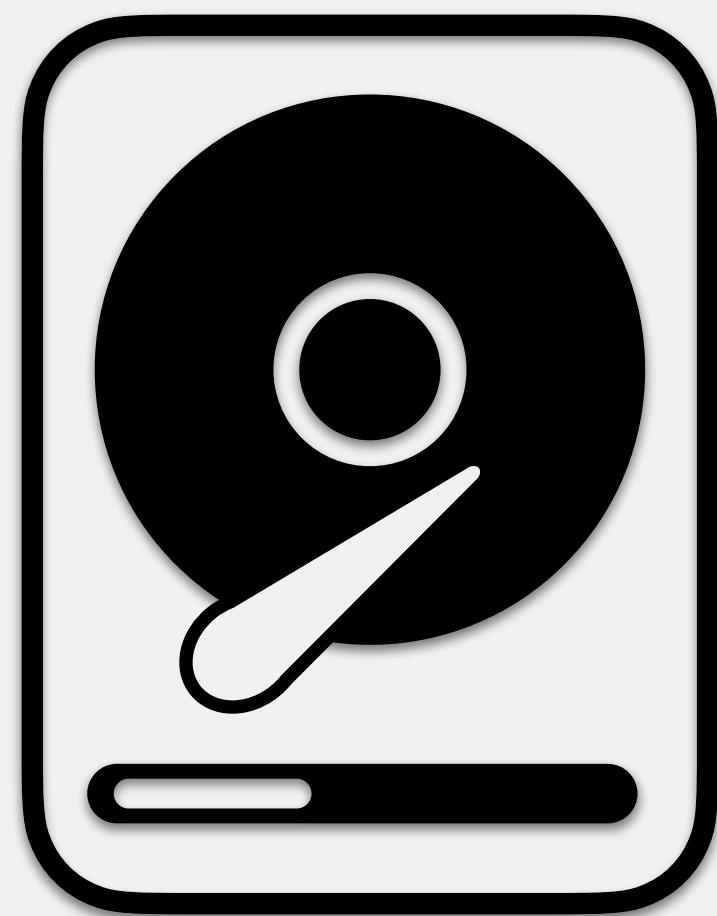
        String files[] = srcDir.list();

        try {
            for (String file : files) {
                File sFile = new File(src, file);
                File dFile = new File(dest, file);
            }
        }
    }
}
```

DataInputStream/FileInputStream

Java classes used together to read **binary** files

cannot read text files!



FileInputStream

DataInputStream

reads data
from file

allows the programmer to
interpret data as primitives

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.Scanner;

public class FileManipulation {

    public static void main(String[] args) {

        /**
         * Copies the .txt files from the dir
         */
        private static void copyFiles() {

            File srcDir = new File("files/");
            File destDir = new File("./");

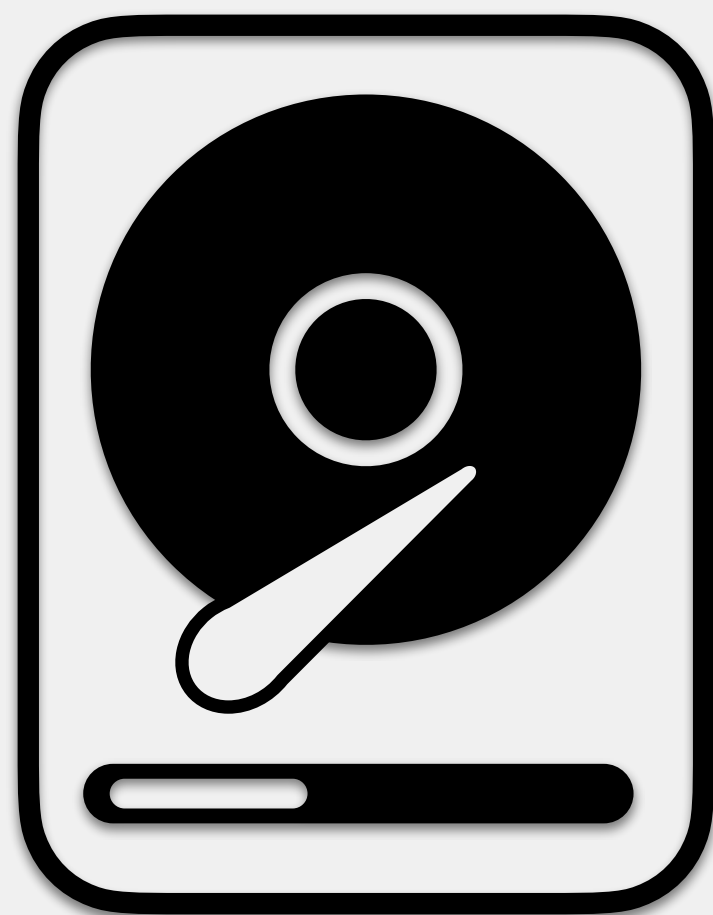
            String files[] = srcDir.list();

            try {
                for (String file : files) {
                    File sFile = new File(src, file);
                    File dFile = new File(dest, file);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

DataOutputStream/FileOutputStream

Java classes used together to write to **binary** files

cannot write to text files!



FileOutputStream DataOutputStream

writes data
to file

programmer-friendly
write methods

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.Scanner;

public class FileManipulation {

    public static void main(String[] args) {
    }

    /**
     * Copies the .txt files from the dir
     */
    private static void copyFiles() {

        File srcDir = new File("files/");
        File destDir = new File("./");

        String files[] = srcDir.list();

        try {
            for (String file : files) {
                File sFile = new File(src, file);
                File dFile = new File(dest, file);
            }
        }
    }
}
```