Subtleties of conditional execution

reading: 4.3

Comparing objects

What is the output?

```
Scanner console = new Scanner(System.in);
  System.out.print("What is your name? ");
  String name = console.next();
  if (name == "Barney") {
      System.out.println("I love you, you love me,");
      System.out.println("We're a happy family!");
  }
```

- Relational operators such as `<` and `==` only behave correctly on primitive values - not objects.
  - The `==` operator on `String`s often evaluates to `false` even when the `String`s have the same letters in them.

The equals method

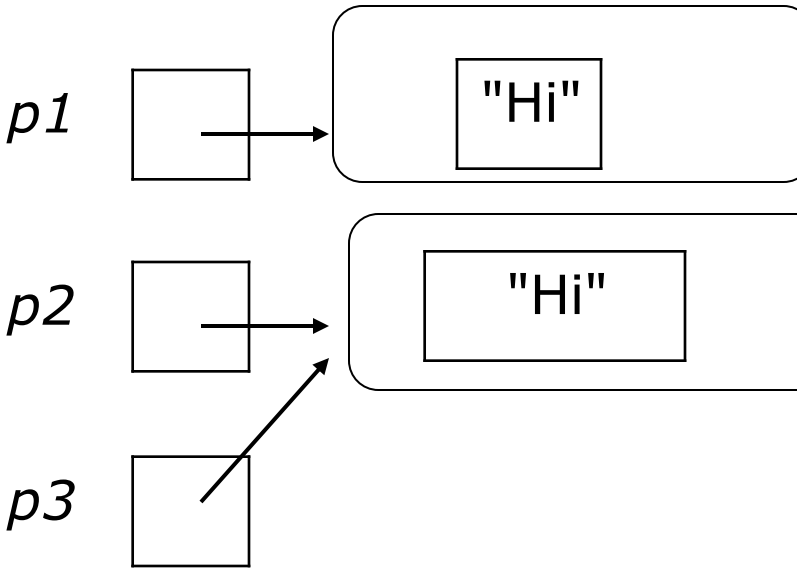
- Objects (such as `String`, `Point`, and `Color`) should be compared for equality by calling a method named `equals`.
  - Example (correct):

```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
if (name.equals("Barney")) {
    System.out.println("I love you, you love me,");
    System.out.println("We're a happy family!");
}
```

Another example

- The `==` operator on objects actually compares whether two variables refer to the same object.
- The `equals` method compares whether two objects have the same state as each other.
  - Given the following code:

```
String p1 = new String("Hi");
String p2 = new String("Hi");
String p3 = p2;
```



  - What is printed?

```
if (p1 == p2) {
    System.out.println("1");
}
if (p1.equals(p2)) {
    System.out.println("2");
}
if (p2 == p3) {
    System.out.println("3");
}
```

String condition methods

- There are several methods of a `String` object that can be used as conditions in `if` statements:

Method	Description
`equals(`*`str`*`)`	whether two strings contain exactly the same characters
`equalsIgnoreCase(`*`str`*`)`	whether two strings contain the same characters, ignoring upper vs. lower case differences
`startsWith(`*`str`*`)`	whether one string contains the other's characters at its start
`endsWith(`*`str`*`)`	whether one string contains the other's characters at its end

String condition examples

- Hypothetical examples, assuming the existence of various `String` variables:

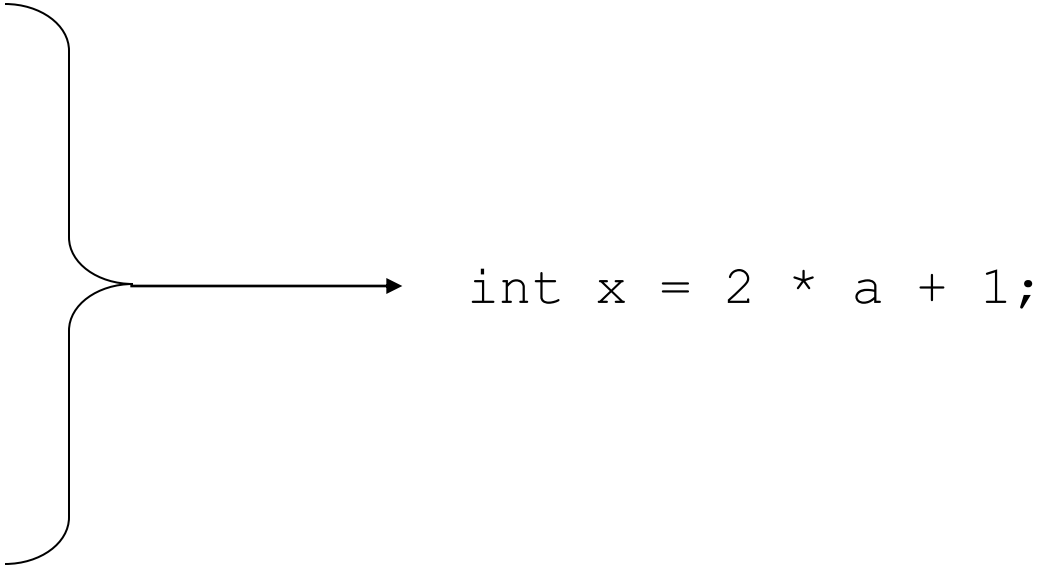
```
if (title.endsWith("Ph. D.")) {
      System.out.println("How's life in the ivory tower?");
  }

if (fullName.startsWith("Queen")) {
      System.out.println("Greetings, your majesty.");
  }

if (name.toLowerCase().indexOf("jr.") >= 0) {
      System.out.println("You share your parent's name.");
  }
```

Factoring if/else code

- **factoring**: extracting common/redundant code
  - Factoring `if/else` code reduces the size of the `if` and `else` statements and can sometimes eliminate the need for `if/else` altogether.

- Example:

```
int x;
if (a == 1) {
    x = 3;
} else if (a == 2) {
    x = 5;
} else {  // a == 3
    x = 7;
}
```



```
int x = 2 * a + 1;
```

Code in need of factoring

- The following example has a lot of redundant code:

```
if (money < 500) {
    System.out.println("You have, $" + money + " left.");
    System.out.print("Caution!  Bet carefully.");
    System.out.print("How much do you want to bet? ");
    bet = console.nextInt();
} else if (money < 1000) {
    System.out.println("You have, $" + money + " left.");
    System.out.print("Consider betting moderately.");
    System.out.print("How much do you want to bet? ");
    bet = console.nextInt();
} else {
    System.out.println("You have, $" + money + " left.");
    System.out.print("You may bet liberally.");
    System.out.print("How much do you want to bet? ");
    bet = console.nextInt();
}
```

Code after factoring

- Factoring tips:
  - If the start of each branch is the same, move it *before* the `if/else`.
  - If the end of each branch is the same, move it *after* the `if/else`.

```
System.out.println("You have, $" + money + " left.");

if (money < 500) {
    System.out.print("Caution!  Bet carefully.");
} else if (money < 1000) {
    System.out.print("Consider betting moderately.");
} else {
    System.out.print("You may bet liberally.");
}

System.out.print("How much do you want to bet? ");
bet = console.nextInt();
```

Parameter question

- Rewrite the following program to use parameterized methods:

```
// Draws triangular figures of stars.
public class Loops {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; i++) {
            for (int j = 1; j <= i - 1; j++) {
                System.out.print(" ");
            }
            for (int j = 1; j <= 10 - 2 * i + 1; j++) {
                System.out.print("*");
            }
            System.out.println();
        }
        for (int i = 1; i <= 12; i++) {
            for (int j = 1; j <= i - 1; j++) {
                System.out.print(" ");
            }
            for (int j = 1; j <= 25 - 2 * i; j++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```

Fencepost loops

textbook section: 4.1

The fencepost problem

- Problem: Write a static method named `printNumbers` that prints each number from 1 to a given maximum, separated by commas.

  For example, the method call:

  ```
  printNumbers(5)
  ```

  should print:

  ```
  1, 2, 3, 4, 5
  ```

Flawed solution 1

- A flawed solution:

```
public static void printNumbers(int max) {
    for (int i = 1; i < max; i++) {
        System.out.print(i + ", ");
    }

}
```

- Output from `printNumbers(5)`:

```
1, 2, 3, 4, 5,
```

Flawed solution 2

- Another flawed solution:

```
public static void printNumbers(int max) {
    for (int i = 1; i <= max; i++) {
        System.out.print(", " + i);
    }
    System.out.println();  // to end the line of output
}
```

- Output from `printNumbers(5)`:

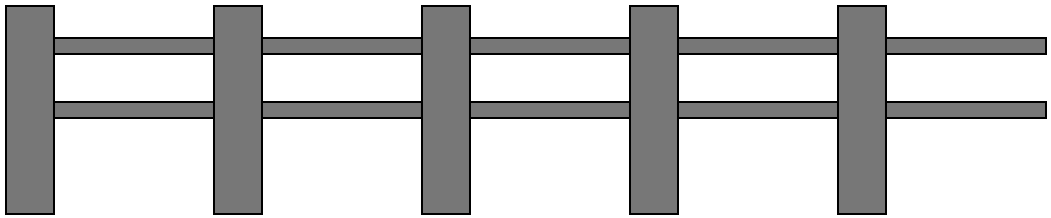
```
, 1, 2, 3, 4, 5
```

Fence post analogy

- We print *n* numbers but need only *n* - 1 commas.
- This problem is similar to the task of building a fence with lengths of wire separated by posts.
  - often called a *fencepost problem*
  - If we repeatedly place a post and wire, the last post will have an extra dangling wire.

  - A flawed algorithm:

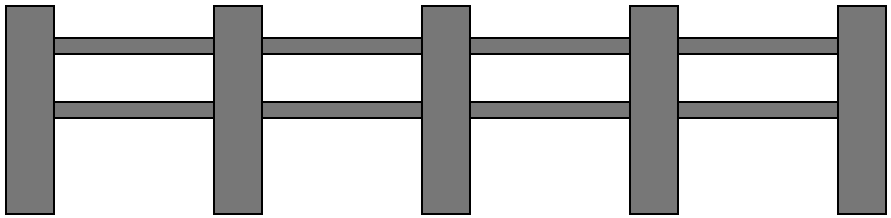
```
for (length of fence) {
    place some post.
    place some wire.
}
```



Fencepost loop

- The solution is to add an extra statement outside the loop that places the inital "post."
  - This is sometimes also called a *fencepost loop* or a "loop-and-a-half" solution.
  - The revised algorithm:

    place a post.
    *for (length of fence **- 1**) {*
        place some wire.
        place some post.
    }



Fencepost method solution

- A version of `printNumbers` that works:

```
public static void printNumbers(int max) {
    System.out.print(1);
    for (int i = 2; i <= max; i++) {
        System.out.print(", " + i);
    }
    System.out.println();  // to end the line of output
}
```

OUTPUT from `printNumbers(5)`:

```
1, 2, 3, 4, 5
```

Fencepost question

- Write a method named `printFactors` that, when given a number, prints its factors in the following format (using an example of 24 for the parameter value):

```
[1, 2, 3, 4, 6, 8, 12, 24]
```

Fencepost question

- Write a Java program that reads a base and a maximum power and prints all of the powers of the given base up to that max, separated by commas.

```
Base: 2
Max exponent: 9

The first 9 powers of 2 are:
2, 4, 8, 16, 32, 64, 128, 256, 512
```

Objects as method parameters: value vs. reference semantics

reading: 3.3

String Objects

- **string**: A sequence of text characters.
  - One of the most common types of objects.
  - In Java, strings are represented as objects of class `String`.

String objects

- `String` variables can be declared and assigned, just like primitive values:

```
String <name> = "<text>";
String <name> = <expression that produces a String>;
```

  - Unlike most other objects, a `String` does not have to be created with `new`.

  - Examples:

```
String name = "Marla Singer";

int x = 3, y = 5;
String point = "(" + x + ", " + y + ")";
```

Indexes

- The characters in a `String` are each internally numbered with an *index*, starting with 0:
  - Example:

```
String name = "P. Diddy";
```

index	0	1	2	3	4	5	6	7
character	`'P'`	`'.'`	`' '`	`'D'`	`'i'`	`'d'`	`'d'`	`'y'`

- Individual characters are represented inside the `String` by values of a primitive type called `char`.
  - Literal `char` values are surrounded with apostrophe (single-quote) marks, such as `'a'` or `'4'`.
  - An escape sequence can be represented as a `char`, such as `'\n'` (new-line character) or `'\''` (apostrophe).

String methods

- Useful methods of each `String` object:

Method name	Description
`charAt(`*index*`)`	character at a specific index
`indexOf(`*str*`)`	index where the start of the given string appears in this string (-1 if it is not there)
`length()`	number of characters in this string
`substring(`*index1*`, `*index2*`)`	the characters in this string from *index1* (inclusive) to *index2* (exclusive)
`toLowerCase()`	a new string with all lowercase letters
`toUpperCase()`	a new string with all uppercase letters

- These methods are called using the dot notation:

```
String example = "speak friend and enter";
System.out.println(example.toUpperCase());
```

String method examples

```
//      index 012345678901
String s1 = "Yvonne Coady";
String s2 = "LillAnne Who?";
System.out.println(s1.length());          // 12
System.out.println(s1.indexOf("e"));      // 5
System.out.println(s1.substring(1, 4));   // von

String s3 = s2.toUpperCase();
System.out.println(s3.substring(6, 10)); // NE W

String s4 = s1.substring(0, 6);
System.out.println(s4.toLowerCase());   // yvonne
```

Modifying Strings

- The methods that appear to modify a string (`substring`, `toLowerCase`, `toUpperCase`, etc.) actually create and return a new string.

```
String s = "lil bow wow";
s.toUpperCase();
System.out.println(s);   // output: lil bow wow
```

- If you want to modify the variable, you must reassign it to store the result of the method call:

```
String s = "lil bow wow";
s = s.toUpperCase();
System.out.println(s);   // output: LIL BOW WOW
```

String methods

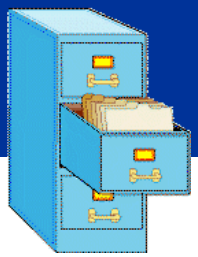
- Given the following string:

```
String book = "Building Java Programs";
```

  - How would you extract the word `"Java"` ?
  - How would you change `book` to store:
    `"BUILDING JAVA PROGRAMS"` ?
  - How would you extract the first word from any general string?

Method name	Description
`charAt(`*index*`)`	character at a specific index
`indexOf(`*str*`)`	index where the start of the given string appears in this string (-1 if it is not there)
`length()`	number of characters in this string
`substring(`*index1*`, `*index2*`)`	the characters in this string from *index1* (inclusive) to *index2* (exclusive)
`toLowerCase()`	a new string with all lowercase letters
`toUpperCase()`	a new string with all uppercase letters

I/O and Files



- Programmers refer to input/output as "I/O".
- file input using `Scanner`
  - `File` objects
  - throwing exceptions
  - file names and folder paths
  - token-based file processing
- complex I/O
  - handling the file-not-found case
  - file output using `PrintStream`

Scanners to read files

- To read a file, create a `File` object and pass it as a parameter when constructing a `Scanner`.

- Creating a `Scanner` for a file, general syntax:

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

Example:

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

or:

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

File and path names

- **relative path**: does not specify any top-level folder
  - `"names.dat"`
  - `"input/kinglear.txt"`
- **absolute path**: specifies drive letter or top `"/"` folder
  - `"C:/Documents/smith/hw6/input/data.csv"`
  - Windows systems also use backslashes to separate folders. How would the above filename be written using backslashes?

- When you construct a `File` object with a relative path, Java assumes it is relative to the *current directory*.
  - `Scanner input = new Scanner(new File("data/readme.txt"));`
  - If our program is in: `H:/johnson/hw6`,
    Java will look for: `H:/johnson/hw6/data/readme.txt`.

Compiler error with files

- The following program does not compile:

```
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"));
        String text = input.next();
        System.out.println(text);
    }
}
```

- The following compiler error is produced:

```
ReadFile.java:6: unreported exception
java.io.FileNotFoundException; must be caught or declared
to be thrown
        Scanner input = new Scanner(new File("data.txt"));
                        ^
```

Exceptions

- **exception**: An object that represents a program error.
  - Programs that contain invalid logic cause ("*throw*") exceptions.
  - Trying to read a file that does not exist will throw an exception.

- **checked exception**: An error that Java forces us to handle in our program (otherwise it will not compile).
  - We must specify what our program will do to handle any potential file I/O failures.
  - We must either:
    - declare that our program will handle ("*catch*") the exception, or
    - explicitly state that we choose not to handle the exception (and we accept that our program will crash if an exception occurs)

Throwing exception syntax

- **`throws` clause**: Keywords on a method's header to state that it may throw an exception.
  - Somewhat like a waiver of liability form:
    "I hereby agree that this method might throw an exception, and I accept the consequences (crashing) if this happens."

- throws clause, general syntax:

```
public static <type> <name>(<params>) throws <type> {
```

  - When doing file I/O, we use `FileNotFoundException`.

```
public static void main(String[] args)
        throws FileNotFoundException {
```

Fixed compiler error

The following corrected program does compile:

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

public class ReadFile {
  public static void main(String[] args)
              throws FileNotFoundException {
    Scanner input = new Scanner(new File("data.txt"));
    String text = input.next();
    System.out.println(text);
  }
}
```

Files and input cursor

- Consider a file `numbers.txt` 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\n\n3.9 4.7 -15.4\n2.8\n`
    `^`
  - When you call the methods of the `Scanner` such as `next` or `nextDouble`, the `Scanner` breaks apart the input into *tokens*.

Input tokens

- **token**: A unit of user input. Tokens are separated by whitespace (spaces, tabs, new lines).

- Example: If an input file contains the following:

```
23   3.14
  "John Smith"
```

  - The tokens in the input are the following, and can be interpreted as the given types:

Token	Type(s)
1. `23`	`int`, `double`, `String`
2. `3.14`	`double`, `String`
3. `"John`	`String`
4. `Smith"`	`String`

Consuming tokens

- Each call to `next`, `nextInt`, `nextDouble`, etc. advances the cursor to the end of the current token, skipping over any whitespace.
  - We call this *consuming* input.

```
input.nextDouble()
```

- **308.2**\n 14.9 7.4 2.8\n\n\n3.9 4.7 -15.4\n2.8\n
  ^

```
input.nextDouble()
```

- 308.2\n **14.9** 7.4 2.8\n\n\n3.9 4.7 -15.4\n2.8\n
  ^

File input question

- Consider an input file named `numbers.txt` that contains the following text:

```
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. Its output:

```
number = 308.2
number = 14.9
number = 7.4
number = 2.8
number = 3.9
Sum = 337.19999999999993
```

File input answer

```
// 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.*;

public class Echo {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers.txt"));
        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);
    }
}
```

Testing before reading

- The preceding program is impractical because it only processes exactly 5 values from the input file.
  - A better program would read the entire file, regardless of how many values it contains.

- The `Scanner` has useful methods for testing to see what the next input token will be:

Method Name	Description
`hasNext()`	whether any more tokens remain
`hasNextDouble()`	whether the next token can be interpreted as type `double`
`hasNextInt()`	whether the next token can be interpreted as type `int`
`hasNextLine()`	whether any more lines remain

File input question

- Consider an input file named `numbers.txt` that contains the following text:

```
308.2
   14.9 7.4  2.8

3.9 4.7    -15.4
    2.8
```

- Write a program that reads the values from this file and prints them along with their sum. Its output:

```
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
```

Test before read answer

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

import java.io.*;
import java.util.*;

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);
    }
}
```

Line-by-line processing

- `Scanner` has a method `nextLine` that returns from the input cursor's position to the nearest `\n` character.
  - You can use `nextLine` to break up a file's contents by line and examine each line individually.

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

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

Line-based input

- `nextLine` consumes and returns a line as a `String`.
  - The `Scanner` moves its cursor until it sees a `\n` new line character and returns the text found.
    - The `\n` character is consumed but not returned.
    - `nextLine` is the only non-token-based `Scanner` method.
    - Recall that the `Scanner` also has a `hasNextLine` method.
  - Example:

```
23   3.14 John Smith   "Hello world"
          45.2      19

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

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

File processing question

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

```
Kelly,

Can you please modify the a5/turnin settings
to make CSE 142 Homework 5 due Wednesday,
July 27 at 11:59pm instead of due tomorrow
at 6pm?

Thanks, Joe
```

- Example output:

```
> Kelly,
>
> Can you please modify the a5/turnin settings
> to make CSE 142 Homework 5 due Wednesday,
> July 27 at 11:59pm instead of due tomorrow
> at 6pm?
>
> Thanks, Joe
```

File processing answer

```
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);
    }
  }
}
```

Processing tokens of one line

- Many input files have a data record on each line.
  - The contents of each line contain meaningful tokens.
  - 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.

```
Enter a name: Brad
Brad (ID#456) worked 36.8 hours (7.36 hours/day)
```

  - Neither line-based nor token-based processing is quite right.
  - The better solution is a hybrid approach in which we break the input into lines, and then break each line into tokens.

Scanners on Strings

- 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
```

- We can use this idea to tokenize each line of a file.

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

Line processing example

- Example: Read in a file containing HTML text, and surround all-uppercase tokens with < and > .
  - Retain the original orientation of the tokens on each line.

Input file:	Output to console:
HTML HEAD TITLE My web page /TITLE /HEAD BODY P There are pics of my cat here, as well as my B cool /B blog, with pics from my Vegas trip. /BODY /HTML	<HTML> <HEAD> <TITLE> My web page </TITLE> </HEAD> <BODY> <P> There are pics of my cat here, as well as my <B> cool </B> blog, stuff about my Vegas trip. </BODY> </HTML>

```
Scanner input = new Scanner(new File("page.html"));
while (input.hasNextLine()) {
    String line = input.nextLine();
    Scanner lineScan = new Scanner(line);
    while (lineScan.hasNext()) {
        String token = lineScan.next();
        if (token.equals(token.toUpperCase())) {   // an HTML tag
            System.out.print("<" + token + "> ");
        } else {
            System.out.print(token + " " );
        }
    }
    System.out.println();
}
```

Prompting for a file name

- We can ask the user to tell us the file to read.
  - Use the `nextLine` method on the console `Scanner`, because the file name might have spaces in it.

```
// prompt for the file name
Scanner console = new Scanner(System.in);
System.out.print("Type a file name to use: ");
String filename = console.nextLine();

Scanner input = new Scanner(new File(filename));
```

- What if the user types a file name that does not exist?

Fixing file-not-found issues

- `File` objects have an `exists` method:

```
Scanner console = new Scanner(System.in);
System.out.print("Type a file name to use: ");
String filename = console.nextLine();
File file = new File(filename);

while (!file.exists()) {
    System.out.print("File not found! Try again: ");
    String filename = console.nextLine();
    file = new File(filename);
}
Scanner input = new Scanner(file);  // open the file
```

```
Type a file name to use: hourz.text
File not found! Try again: h0urz.txt
File not found! Try again: hours.txt
```

Output to files

- **`PrintStream`**: An object in the `java.io` package that lets you print output to a destination such as a file.
  - `System.out` is also a `PrintStream`.
  - Any methods you have used on `System.out` (such as `print`, `println`) will work on every `PrintStream`.

- Printing into an output file, general syntax:

```
PrintStream <name> =
        new PrintStream(new File("<file name>"));
...
```

  - If the given file does not exist, it is created.
  - If the given file already exists, it is overwritten.

Printing to files, example

- Example:

```
PrintStream output = new PrintStream(new File("output.txt"));
output.println("Hello, file!");
output.println("This is a second line of output.");
```

  - You can use similar ideas about prompting for file names here.

- Do not open a file for reading (`Scanner`) and writing (`PrintStream`) at the same time.
  - The result can be an empty file (size 0 bytes).
  - You could overwrite your input file by accident!