Principles of Computer Science I

Prof. Nadeem Abdul Hamid

CSC 120A - Fall 2004

Lecture Unit 7

Review Chapter 4

- Boolean data type and operators (&&, ||,…)
- Selection control flow structure
  - *if*, *if-else*, nested *if* structures
- Testing, algorithm walk-though, execution trace
- Encapsulation and abstraction

File Input/Output

- Up till now, we have been interacting with our programs through screen and keyboard
- It is also useful to be able to input and output data using a file on disk instead
- Using a file for input allows us to:
  - Handle large quantities of data
  - Type a little bit at a time
  - Go back and fix mistakes
  - Can re-run the program with the same data without having to retype it
- Using a file for output we can
  - View output on the screen or print it
  - Examine output without having to re-run the program
  - Store data that is used as input for another program

Five Steps to File IO

1. Import Java library: java.io.*
2. Declare file variable identifier
3. Instantiate file object and assign to the file variable
4. Use methods of the file object to read or write data
5. Call a method to close the file when we are done

File IO: Step 1

```
import java.io.*;
```

- We already know how to do that
- For files, we will be using the *FileReader*, *FileWriter*, *BufferedReader*, and *PrintWriter* classes from the library
  - FileReader and FileWriter provide basic functionality of reading/writing one character at a time
  - PrintWriter allows us to output data to a file just like we've been outputting data to the screen with the System.out object

File IO: Step 2

```
PrintWriter outFile;
BufferedReader inFile;
```

- Declare file identifiers like any other variable
- BufferedReader for input files, PrintWriter for output files
  - These classes work with *character stream files* (files that we view and change in a text editor)
    - Data is organized in lines (sequences of characters)
    - Each line ends with an EOL (end-of-line) mark that editor doesn't display – it goes to the next line as it places characters on the screen

File IO: Step 3

```
outFile = new PrintWriter(new FileWriter(“outFile.txt”));
inFile = new BufferedReader(new FileReader(“inFile.txt”));
```

- Create file objects for use in your program and associate them with physical files on the disk
- With input file: *file pointer* is placed at the first character in the file
- With output file: *creates* a new empty file, or *erases* old contents of existing file

File IO: Step 4 (Output)

```
outFile.print(“The answer is “ + 49);
outFile.println(“Rate = “ + rate);
```

- Just like System.out.print and println
- println() adds an EOL mark to the end of the string as it is saved in the file

File IO: Step 4 (Input)

```
String line = inFile.readLine();
int num = Integer.parseInt(inFile.readLine());
```

- Exactly like reading data from the keyboard, because we are using the same BufferedReader class
- readLine() discards the EOL mark as it is reading a line of characters from the file
  - returns null if we’ve reached the end of file (EOF)
- We can skip over a bunch of letters in a file:
  inFile.skip(100L); // L means it’s a long integer literal
- Throws an exception if we try to skip past the end of the file
  - Need to include “throws IOException” clause after methods that use these input methods

File IO: Step 5

```
inFile.close();
outFile.close();
```

- Breaks the association between the physical file and the variable (inFile/outFile)
- Makes file available for use by other programs
- Be nice: close files once you are done with them
- A simple program using files… (UseFile.java)

Looping

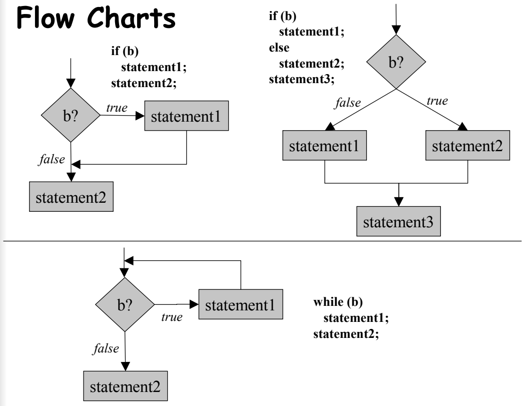
- Control structure that causes a statement or group of statements to be executed repeatedly
- Pattern:
  ```
  while ( <boolean-expression> )
      <statement>
  ```
- Example:
  ```
  while ( count <= 25 )
     count = count + 1;
  ```
- Body of a loop can be single statement or a group of statements enclosed in { … }

Flow Charts

```
if (b)
   statement1;
statement2;
```

```
if (b)
   statement1;
else
   statement2;
statement3;
```

```
while (b)
   statement1;
statement2;
```



Loop Terminology

- Loop entry: point at which flow of control reaches first statement inside a loop
- Iteration: an individual pass through, or repetition of, the body of a loop
- Loop test: point at which *while* expression is evaluated to decide whether to loop or not
- Loop exit: point at which control passes to first statement after the loop body
- Termination condition: condition that causes loop to be exited

Types of Loops

- Count-controlled loop: a loop that executes a specified number of times
- Event-controlled loop: loop that terminates when something happens inside the body to signal that the loop should be exited

- Making an angel food cake:
  - "Beat the mixture 300 strokes" (count-controlled loop)
  - "Cut with a pastry blender until the mixture resembles coarse meal" (event-controlled loop)

Count-Controlled Loops

- Use a variable (*loop control variable*) in the test
- Before entering the loop, must *initialize* the loop control variable
- In each iteration of the loop, must *update* (usually *increment by 1*) the loop control variable

```
int loopCount = 1;
while (loopCount <= 10) {
    …
    loopCount = loopCount + 1;
}
```

Example: Count-Controlled Loop

```
int mult = 1;
while (mult <= 10) {
   System.out.println("2 times " + mult + " is " +
                                   (2 * mult));
   mult++;       // same as: mult = mult + 1;
}
```

- Redo BinaryConv.java using a loop…
- If you forget to properly initialize or update the loop control variable your program will go into the famous *infinite loop*.

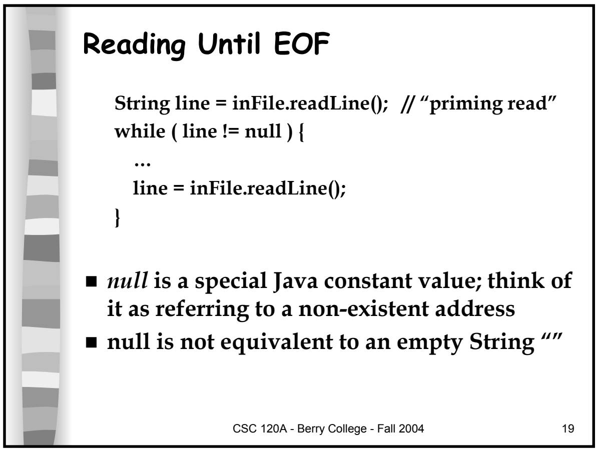
Event-Controlled Loops

- Loops often used to read in and process long lists of data
  - Amount of data is unknown so we cannot use a count-controlled loop
- Instead, we read/process data until some special data value is reached, or until the end of file
- A *sentinel* (or *trailer*) *value* in a file is used as a signal that the end of data to be processed has been reached
  - E.g. In a program that reads in a calendar dates, we may use the date February 31 as a sentinel

Loop with a Sentinel Value

```
String date = inFile.readLine();  // "priming read"
while ( !date.equals("0231") ) {
   …
   date = inFile.readLine();
}
```

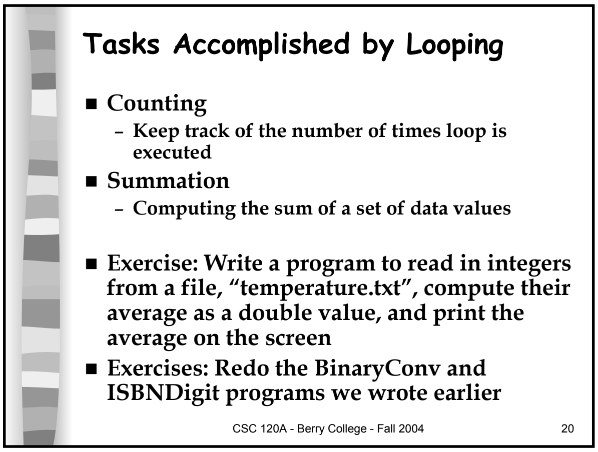
- Priming read: before entering the loop, we must read the first data value
- At the end of the loop body, read in the next data value



Reading Until EOF

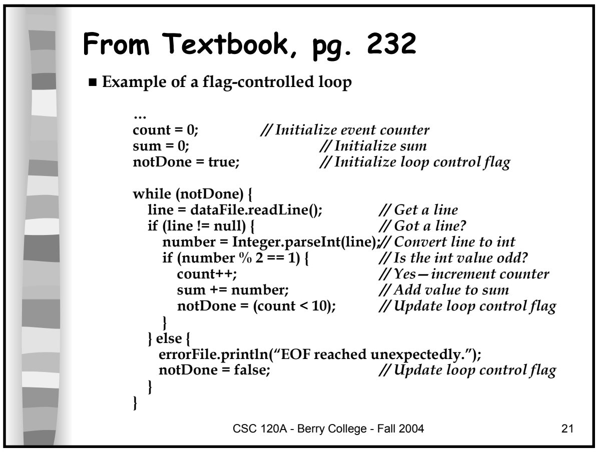
```
String line = inFile.readLine();  // "priming read"
while ( line != null ) {
   …
   line = inFile.readLine();
}
```

- *null* is a special Java constant value; think of it as referring to a non-existent address
- null is not equivalent to an empty String ""



Tasks Accomplished by Looping

- Counting
  - Keep track of the number of times loop is executed
- Summation
  - Computing the sum of a set of data values
- Exercise: Write a program to read in integers from a file, "temperature.txt", compute their average as a double value, and print the average on the screen
- Exercises: Redo the BinaryConv and ISBNDigit programs we wrote earlier

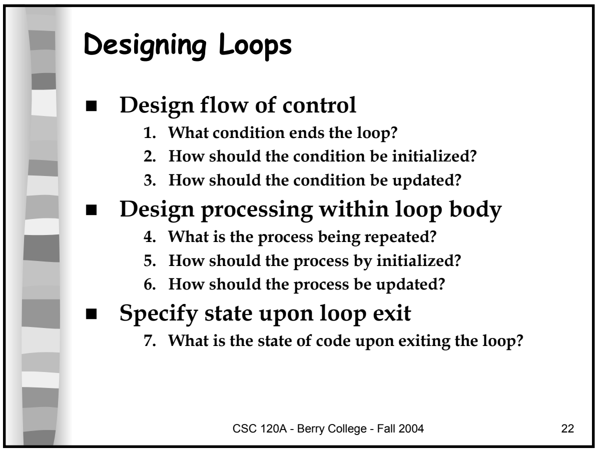


From Textbook, pg. 232

- Example of a flag-controlled loop

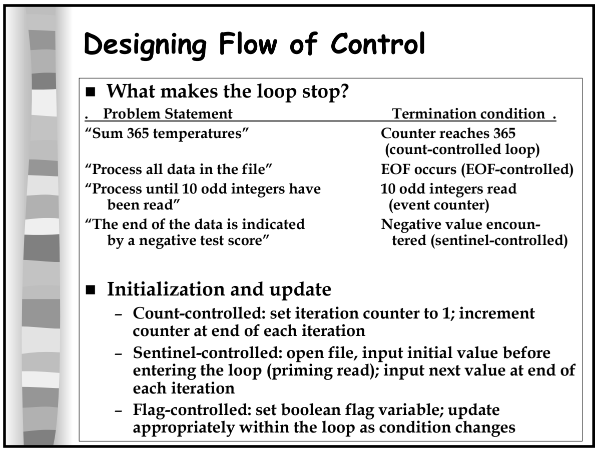
```
…
count = 0;              // Initialize event counter
sum = 0;                          // Initialize sum
notDone = true;                   // Initialize loop control flag

while (notDone) {
   line = dataFile.readLine();            // Get a line
   if (line != null) {                    // Got a line?
      number = Integer.parseInt(line);// Convert line to int
      if (number % 2 == 1) {              // Is the int value odd?
         count++;                         // Yes—increment counter
         sum += number;                   // Add value to sum
         notDone = (count < 10);          // Update loop control flag
      }
   } else {
      errorFile.println("EOF reached unexpectedly.");
      notDone = false;                    // Update loop control flag
   }
}
```



Designing Loops

- Design flow of control
  1. What condition ends the loop?
  2. How should the condition be initialized?
  3. How should the condition be updated?
- Design processing within loop body
  4. What is the process being repeated?
  5. How should the process by initialized?
  6. How should the process be updated?
- Specify state upon loop exit
  7. What is the state of code upon exiting the loop?

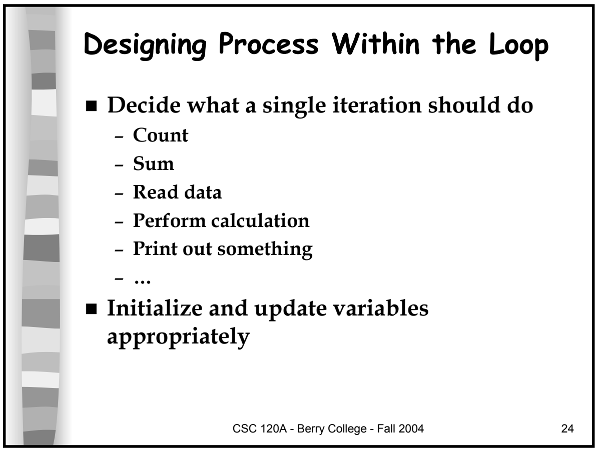


Designing Flow of Control

- What makes the loop stop?

. Problem Statement	Termination condition .
"Sum 365 temperatures"	Counter reaches 365 (count-controlled loop)
"Process all data in the file"	EOF occurs (EOF-controlled)
"Process until 10 odd integers have been read"	10 odd integers read (event counter)
"The end of the data is indicated by a negative test score"	Negative value encoun-tered (sentinel-controlled)

- Initialization and update
  - Count-controlled: set iteration counter to 1; increment counter at end of each iteration
  - Sentinel-controlled: open file, input initial value before entering the loop (priming read); input next value at end of each iteration
  - Flag-controlled: set boolean flag variable; update appropriately within the loop as condition changes



Designing Process Within the Loop

- Decide what a single iteration should do
  - Count
  - Sum
  - Read data
  - Perform calculation
  - Print out something
  - …
- Initialize and update variables appropriately

Loop Exit

- Check the condition of variables upon loop exit (especially check for off-by-one errors)

```
lineCount = 1;
while ((inLine = inFile.readLine()) != null)
   lineCount++;
System.out.println("There are " + lineCount +
                                   " lines in the file.");
```

  – *(above code is incorrect)*

Nested Loops

- Create more complex and useful control structures (just like *if* statements)

```
Initialize outer loop
while ( Outer-loop-condition ) {
      ...
      Initialize inner loop
      while ( Inner-loop-condition ) {
         Inner loop processing
            and update
      }
      ...
}
```

Example: Counting Commas in a File

- Partial program on page 236-237
  - Design loops using the seven steps on slide 22
  - Use the *charAt(n)* method of the String class, which returns the character at a given position in the string ("ABCDE".charAt(0) returns 'A')
- Exercise: How would you implement the *MakeSpaces.spaces(n)* method that we used in lab?

```
public String spaces(int n) { …
```

Loop Testing and Debugging

- Develop test data for loops to check all possible scenarios
  - Loop is skipped entirely
  - Loop body executed exactly once
  - Loop executes a normal number of times
  - Loop fails to exit
- Check loop termination condition carefully
- Watch out for "off-by-one" errors
- Trace execution of loop by hand, step by step
- Use debugging output statements to isolate errors

```
System.out.println("count = " + count);
```

  - Can be commented out later

What's Wrong?

- Code segment to print out the even numbers between 1 and 15:

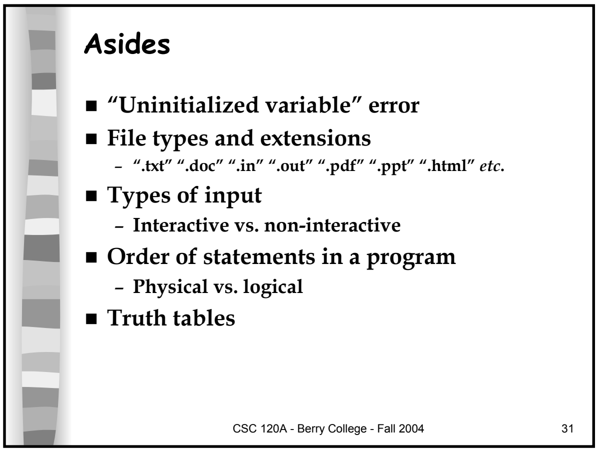
```
int n = 2;
while (n != 15) {
   n = n + 2;
   System.out.print(n + " ");
}
```

- (2 logical errors)

What's Wrong II?

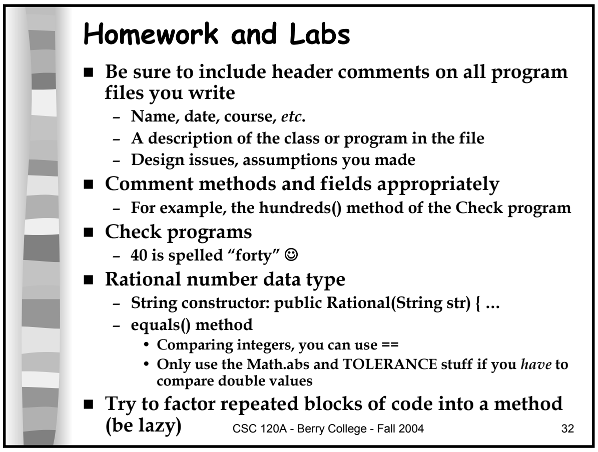
- Code segment to copy a line of text from one file to another, character by character:

```
String line = inFile.readLine();
int count = 1;
while (count < line.length()) {
   outFile.print(line.charAt(count));
   count++;
}
outFile.println();
```



Asides

- "Uninitialized variable" error
- File types and extensions
  - ".txt" ".doc" ".in" ".out" ".pdf" ".ppt" ".html" *etc*.
- Types of input
  - Interactive vs. non-interactive
- Order of statements in a program
  - Physical vs. logical
- Truth tables



Homework and Labs

- Be sure to include header comments on all program files you write
  - Name, date, course, *etc*.
  - A description of the class or program in the file
  - Design issues, assumptions you made
- Comment methods and fields appropriately
  - For example, the hundreds() method of the Check program
- Check programs
  - 40 is spelled "forty" ☺
- Rational number data type
  - String constructor: public Rational(String str) { …
  - equals() method
    - Comparing integers, you can use ==
    - Only use the Math.abs and TOLERANCE stuff if you *have* to compare double values
- Try to factor repeated blocks of code into a method (be lazy)