Building Java Programs

Chapter 5
Lecture 5-1: while Loops, Fencepost Loops, and Sentinel Loops

reading: 5.1 – 5.2

A deceptive problem...

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

For example, the call:

```java
printNumbers(5)
```

should print:

```
1, 2, 3, 4, 5
```
Flawed solutions

- 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,

- 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.
- Similar to building a fence with wires separated by posts:
  - If we use a flawed algorithm that repeatedly places a post + wire, the last post will have an extra dangling wire.

```java
for (length of fence) {
  place a post.
  place some wire.
}
```
Fencepost loop

- Add a statement outside the loop to place the initial "post."
- Also called a fencepost loop or a "loop-and-a-half" solution.

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

Fencepost method solution

```java
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
}
```

- Alternate solution: Either first or last "post" can be taken out:

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

- Modify your method printNumbers into a new method printPrimes that prints all prime numbers up to a max.
  - Example: printPrimes(50) prints
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47
  - If the maximum is less than 2, print no output.

- To help you, write a method countFactors which returns the number of factors of a given integer.
  - countFactors(20) returns 6 due to factors 1, 2, 4, 5, 10, 20.

Fencepost answer

// Prints all prime numbers up to the given max.
public static void printPrimes(int max) {
    if (max >= 2) {
        System.out.print("2");
        for (int i = 3; i <= max; i++) {
            if (countFactors(i) == 2) {
                System.out.print(", " + i);
            }
        }
        System.out.println();
    }
}

// Returns how many factors the given number has.
public static int countFactors(int number) {
    int count = 0;
    for (int i = 1; i <= number; i++) {
        if (number % i == 0) {
            count++;  // i is a factor of number
        }
    }
    return count;
}
while loops

reading: 5.1

Categories of loops

- **definite loop**: Executes a known number of times.
  - The for loops we have seen are definite loops.
    - Print "hello" 10 times.
    - Find all the prime numbers up to an integer \( n \).
    - Print each odd number between 5 and 127.

- **indefinite loop**: One where the number of times its body repeats is not known in advance.
  - Prompt the user until they type a non-negative number.
  - Print random numbers until a prime number is printed.
  - Repeat until the user has typed "q" to quit.
The **while** loop

- **while loop**: Repeatedly executes its body as long as a logical test is true.

  ```java
  while (<test>) {
      <statement(s)>;
  }
  ```

- **Example**:

  ```java
  int num = 1; // initialization
  while (num <= 200) { // test
      System.out.print(num + " ");
      num = num * 2; // update
  }
  // output: 1 2 4 8 16 32 64 128
  ```

---

**Example** **while** loop

- **finds the first factor of 91, other than 1**

  ```java
  int n = 91;
  int factor = 2;
  while (n % factor != 0) {
      factor++;
  }
  System.out.println("First factor is " + factor);
  // output: First factor is 7
  ```

- **while** is better than **for** because we don't know how many times we will need to increment to find the factor.
Sentinel values

- **sentinel**: A value that signals the end of user input.
- **sentinel loop**: Repeats until a sentinel value is seen.

Example: Write a program that prompts the user for text until the user types nothing, then output the total number of characters typed.
- (In this case, the empty string is the sentinel value.)

Type a line (or nothing to exit): hello
Type a line (or nothing to exit): this is a line
Type a line (or nothing to exit): You typed a total of 19 characters.

Solution?

Scanner console = new Scanner(System.in);
int sum = 0;
String response = "dummy"; // "dummy" value, anything but 

while (!response.equals("")) {
    System.out.print("Type a line (or nothing to exit): ");
    response = console.nextLine();
    sum += response.length();
}
System.out.println("You typed a total of " + sum + " characters.");
Changing the sentinel value

• Modify your program to use "quit" as the sentinel value.
  • Example log of execution:

    Type a line (or "quit" to exit): hello
    Type a line (or "quit" to exit): this is a line
    Type a line (or "quit" to exit): quit
    You typed a total of 19 characters.

• Changing the sentinel's value to "quit" does not work!

Scanner console = new Scanner(System.in);
int sum = 0;
String response = "dummy"; // "dummy" value, anything but "quit"

while (!response.equals("quit")) {
    System.out.print("Type a line (or "quit" to exit): ");
    response = console.nextLine();
    sum += response.length();
}

System.out.println("You typed a total of "+ sum + " characters.");

• This solution produces the wrong output. Why?
  You typed a total of 23 characters.
The problem with our code

- Our code uses a pattern like this:
  ```
  sum = 0.
  while (input is not the sentinel) {
    prompt for input; read input.
    add input length to the sum.
  }
  ```

- On the last pass, the sentinel's length (4) is added to the sum:
  ```
  prompt for input; read input ("quit").
  add input length (4) to the sum.
  ```

- This is a fencepost problem.
  - Must read $N$ lines, but only sum the lengths of the first $N$-1.

A fencepost solution

```
sum = 0.
prompt for input; read input. // place a "post"

while (input is not the sentinel) {
  add input length to the sum. // place a "wire"
  prompt for input; read input. // place a "post"
}
```

- Sentinel loops often utilize a fencepost "loop-and-a-half" style solution by pulling some code out of the loop.
Correct code

```java
Scanner console = new Scanner(System.in);
int sum = 0;

// pull one prompt/read ("post") out of the loop
System.out.print("Type a line (or \"quit\" to exit): ");
String response = console.nextLine();

while (!response.equals("quit") ) {
    sum += response.length(); // moved to top of loop
    System.out.print("Type a line (or \"quit\" to exit): ");
    response = console.nextLine();
}

System.out.println("You typed a total of " + sum + " characters.");
```

Sentinel as a constant

```java
public static final String SENTINEL = "quit";
...

Scanner console = new Scanner(System.in);
int sum = 0;

// pull one prompt/read ("post") out of the loop
System.out.print("Type a line (or \" + SENTINEL + \" to exit): ");
String response = console.nextLine();

while (!response.equals(SENTINEL) ) {
    sum += response.length(); // moved to top of loop
    System.out.print("Type a line (or \" + SENTINEL + \" to exit): ");
    response = console.nextLine();
}

System.out.println("You typed a total of " + sum + " characters.");
```