Building Java Programs

Chapter 5
Lecture 11: `while` Loops,
Fencepost Loops, and Sentinel Loops, Assertions

reading: 5.1 – 5.2

(Slides adapted from Stuart Reges, Hélène Martin, and Marty Stepp)

```
while (mahself.stillAwake())
{
     sheep++;
}
```

Methods using `charAt`

- Write a method `printConsonants` that accepts a `String` as a parameter and prints out that `String` with all vowels removed

  For example, the call:

  ```
  printConsonants("atmosphere")
  ```

  should print:

  ```
  tmsphr
  ```

A deceptive problem...

- Write a method `printLetters` that prints each letter from a word separated by commas.

  For example, the call:

  ```
  printLetters("Atmosphere")
  ```

  should print:

  ```
  A, t, m, o, s, p, h, e, r, e
  ```

Flawed solutions

-
  ```
  public static void printLetters(String word) {
      for(int i = 0; i < word.length(); i++) {
          System.out.print(word.charAt(i) + ", ");
      }
      System.out.println();    // end line
  }
  ```

  - Output:
    ```
    A, t, m, o, s, p, h, e, r, e,
    ```

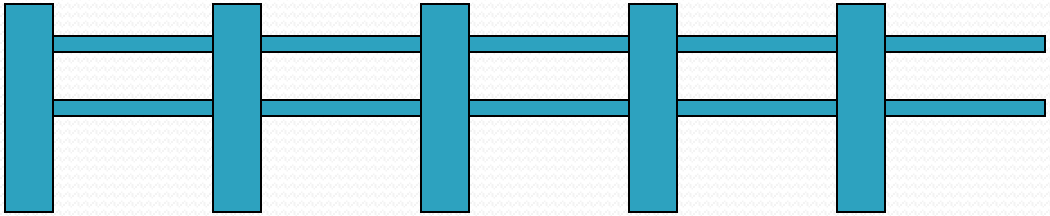
-
  ```
  public static void printLetters(String word) {
      for(int i = 0; i < word.length(); i++) {
          System.out.print(", " + word.charAt(i));
      }
      System.out.println();    // end line
  }
  ```

  - Output:
    ```
    , A, t, m, o, s, p, h, e, r, e
    ```

Fence post analogy

- We print *n* letters 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.

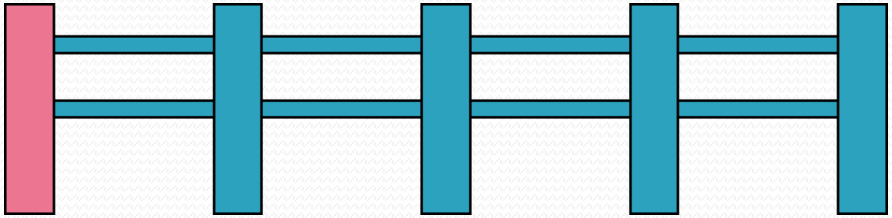
```
for (length of fence) {
    place a post.
    place some wire.
}
```



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.

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



Fencepost method solution

-
```
public static void printLetters(String word) {
    System.out.print(word.charAt(0));
    for(int i = 1; i < word.length(); i++) {
        System.out.print(", " + word.charAt(i));
    }
    System.out.println();    // end line
}
```

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

```
public static void printLetters(String word) {
    for(int i = 0; i < word.length() - 1; i++) {
        System.out.print(word.charAt(i) + ", ");
    }
    int last = word.length() - 1;
    System.out.println(word.charAt(last)); // end line
}
```

Fencepost question

- Write a 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

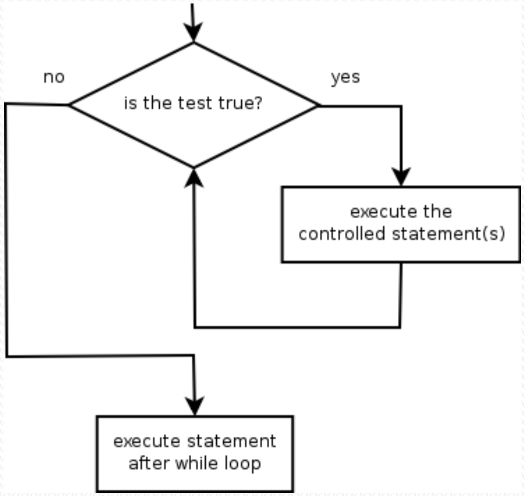
- **definite loop**: Executes a known number of times.
  - The `for` loops we have seen are definite loops.
    - Print "hello" 10 times.
    - Find all the prime numbers up to an integer *n*.
    - Print each odd number between 5 and 127.

- **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.

```
while (test) {
    statement(s);
}
```


no
yes
is the test true?
execute the controlled statement(s)
execute statement after while loop

- Example:

```
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
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 "quit", then output the total number of characters typed.
  - (In this case, "quit" is the sentinel value.)

```
Type a word (or "quit" to exit): hello
Type a word (or "quit" to exit): yay
Type a word (or "quit" to exit): quit
You typed a total of 8 characters.
```

Solution?

```
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 word (or \"quit\" to exit): ");
    response = console.next();
    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 12 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

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

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

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

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

Sentinel as a constant

```
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 word (or \"" + SENTINEL + "\" to exit): ");
String response = console.next();

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

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

Logical assertions

- **assertion**: A statement that is either true or false.

  Examples:
  - Java was created in 1995.
  - The sky is purple.
  - 23 is a prime number.
  - 10 is greater than 20.
  - x divided by 2 equals 7. *(depends on the value of x)*

- An assertion might be false ("The sky is purple" above), but it is still an assertion because it is a true/false statement.

Reasoning about assertions

- Suppose you have the following code:

```
if (x > 3) {
    // Point A
    x--;
} else {
    // Point B
    x++;
    // Point C
}
// Point D
```

- What do you know about `x`'s value at the three points?
    - Is `x > 3`? Always? Sometimes? Never?

Assertions in code

- We can make assertions about our code and ask whether they are true at various points in the code.
  - Valid answers are ALWAYS, NEVER, or SOMETIMES.

```
System.out.print("Type a nonnegative number: ");
double number = console.nextDouble();
// Point A: is number < 0.0 here?        (SOMETIMES)

while (number < 0.0) {
    // Point B: is number < 0.0 here?    (ALWAYS)
    System.out.print("Negative; try again: ");

    number = console.nextDouble();
    // Point C: is number < 0.0 here?    (SOMETIMES)
}

// Point D: is number < 0.0 here?        (NEVER)
```

Reasoning about assertions

- Right after a variable is initialized, its value is known:

```
int x = 3;
// is x > 0?  ALWAYS
```

- In general you know nothing about parameters' values:

```
public static void mystery(int a, int b) {
// is a == 10?  SOMETIMES
```

- But inside an `if`, `while`, etc., you may know something:

```
public static void mystery(int a, int b) {
    if (a < 0) {
        // is a == 10?  NEVER
        ...
    }
}
```

Assertions and loops

- At the start of a loop's body, the loop's test must be `true`:

```
while (y < 10) {
    // is y < 10?  ALWAYS
    ...
}
```

- After a loop, the loop's test must be `false`:

```
while (y < 10) {
    ...
}
// is y < 10?  NEVER
```

- Inside a loop's body, the loop's test may become `false`:

```
while (y < 10) {
    y++;
    // is y < 10?  SOMETIMES
}
```

"Sometimes"

- Things that cause a variable's value to be unknown (often leads to "sometimes" answers):
  - reading from a `Scanner`
  - reading a number from a `Random` object
  - a parameter's initial value to a method

- If you can reach a part of the program both with the answer being "yes" and the answer being "no", then the correct answer is "sometimes".
  - If you're unsure, "Sometimes" is a good guess.

Assertion example 1

```
public static void mystery(int x, int y) {
    int z = 0;

    // Point A

    while (x >= y) {
        // Point B
        x = x - y;
        z++;

        if (x != y) {
            // Point C
            z = z * 2;
        }

        // Point D

    }

    // Point E
    System.out.println(z);
}
```

Which of the following assertions are true at which point(s) in the code? Choose ALWAYS, NEVER, or SOMETIMES.

	x < y	x == y	z == 0
Point A	SOMETIMES	SOMETIMES	ALWAYS
Point B	NEVER	SOMETIMES	SOMETIMES
Point C	SOMETIMES	NEVER	NEVER
Point D	SOMETIMES	SOMETIMES	NEVER
Point E	ALWAYS	NEVER	SOMETIMES