Building Java Programs

Chapter 5:
Program Logic and Indefinite Loops

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

The Big Picture

- Looping is crucially important in most programs
  - knowing the common patterns saves programming time
- Often, the programmer doesn't know how long to loop for
  - most applications soliciting user input
    - game loop
  - web servers

Fencepost loops

reading: 4.1

self-check: 2

exercises: 2, 4, 5, 8

A fencepost problem

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

  For example, the call:

  ```
  printNumbers(5)
  ```

  should print:

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

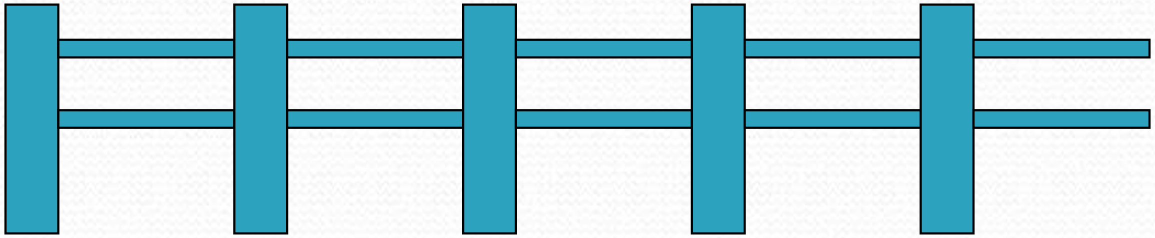
Flawed solutions

- ```
  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 repeatedly place a post+wire,
    the last post will have an extra dangling wire.
  - A flawed algorithm:
    for (length of fence) {
        place a post.
        place some wire.
    }

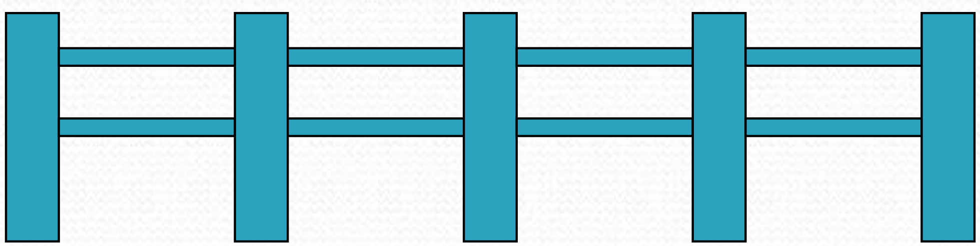


Fencepost loop

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

  - The revised algorithm:

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

Output from `printNumbers(5)`:

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

A second solution

- Either the first or the last "post" can be taken out of the loop:

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

- The output is identical; pick the one that makes most sense to you

Fencepost question

- Write a method `printPrimes` that prints all prime numbers up to a given maximum in the following format.

  - Example: `printPrimes(50)` prints

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

- To find primes, write a method `countFactors` which returns the number of factors an integer has
  - `countFactors(60)` returns 12 because
    1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, and 60 are factors of 60.

Fencepost answer

```
public class Primes {
    public static void main(String[] args) {
        printPrimes(50);
        printPrimes(1000);
    }

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

Fencepost answer, continued

```
// Returns how many factors the given number has.
// Note: this is also in Ch4-1 slides
   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

self-check: 1 - 10

exercises: 1 - 2

Definite loops

- **definite loop**: executes a known number of times.
  - The `for` loops we have seen so far are definite loops.

  - Examples:
    - Print "hello" 10 times.
    - Find all the prime numbers up to an integer *n*.
    - Print each odd number between 5 and 127.

Indefinite loops

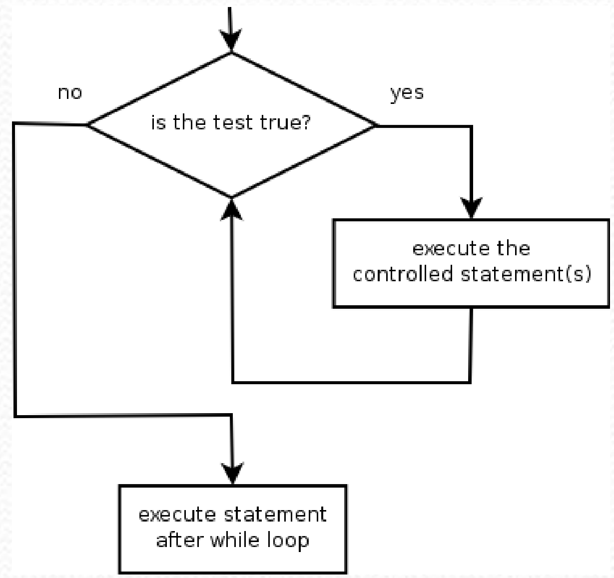
- **indefinite loop**: the number of times its body repeats is not known in advance.
  - The `while` loops we'll see in this chapter are indefinite loops.

  - Examples:
    - Prompt the user until they type a non-negative number.
    - Print random numbers until a prime number is printed.
    - Continue looping while the user has not typed "n" to quit.

The while loop

- **while loop**: Executes as long as a test is true.

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



- 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
```

for vs while loops

- The `for` loop is a specialized form of the `while` loop.
  - Equivalent:

```
for (int num = 1; num <= 200; num = num * 2) {
      System.out.print(num + " ");
}

int num = 1;
while (num <= 200) {
    System.out.print(num + " ");
     num = num * 2;
}
```

- Stylistically, it is better to use a `for` loop when looping over a series of values

Example `while` loop

```
// finds number's first factor other than 1
Scanner console = new Scanner(System.in);
System.out.print("Type a number: ");
int number = console.nextInt();
int factor = 2;
while (number % factor != 0) {
    factor++;
}
System.out.println("First factor: " + factor);
```

- Example log of execution:

```
Type a number: 91
First factor: 7
```

while loop question

- Write code that repeatedly prompts until the user types a non-negative number, then computes its square root.

  - Example log of execution:

```
Type a non-negative integer: -5
Invalid number, try again: -1
Invalid number, try again: -235
Invalid number, try again: -87
Invalid number, try again: 121
The square root of 121 is 11.0
```

while loop answer

- Solution:

```
System.out.print("Type a non-negative integer: ");
int number = console.nextInt();

while (number < 0) {
    System.out.print("Invalid number, try again: ");
    number = console.nextInt();
}

System.out.println("The square root of " + number +
                   " is " + Math.sqrt(number));
```

  - Notice that `number` has to be declared outside the loop.

Sentinel loops

reading: 5.1

self-check: 5
exercises: 1, 2

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 repeatedly prompts the user for numbers until the user types 0, then outputs their sum.

    (In this case, 0 is the sentinel value.)

```
Enter a number (0 to quit): 95
Enter a number (0 to quit): 87
Enter a number (0 to quit): 42
Enter a number (0 to quit): 26
Enter a number (0 to quit): 0
The total is 250
```

Flawed sentinel solution

- What's wrong with this solution?

```
Scanner console = new Scanner(System.in);
int sum = 0;
int number = 1;   // "dummy value", anything but 0

while (number != 0) {
    System.out.print("Enter a number (0 to quit): ");
    number = console.nextInt();
    sum = sum + number;
}

System.out.println("The total is " + sum);
```

A different sentinel value

- Modify your program to use a sentinel value of **-1**.

```
Enter a number (-1 to quit): 95
Enter a number (-1 to quit): 87
Enter a number (-1 to quit): 42
Enter a number (-1 to quit): 26
Enter a number (-1 to quit): -1
The total is 250
```

Changing the sentinel value

- To see the problem, change the sentinel's value to -1:

```
Scanner console = new Scanner(System.in);
int sum = 0;
int number = 1;  // "dummy value", anything but -1

while (number != -1) {
    System.out.print("Enter a number (-1 to quit): ");
    number = console.nextInt();
    sum += number;
}

System.out.println("The total is " + sum);
```

- Now the solution produces the wrong output. Why?

```
The total was 249
```

The problem

- Our code uses a pattern like this:

  sum = 0.
  while (input is not the sentinel) {
  *   prompt for input; read input.*
  *   add input to the sum.*
  }

- On the last pass, the sentinel -1 is added to the sum:

  prompt for input; read input (-1).
  add input (-1) to the sum.

- This is a fencepost problem.
  - We must read *N* numbers, but only sum the first *N*-1 of them.

A fencepost solution

- We need to use a pattern like this:

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

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

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

Correct code

- This solution produces the correct output:

```
Scanner console = new Scanner(System.in);
int sum = 0;
System.out.print("Enter a number (-1 to quit): ");
int number = console.nextInt();

while (number != -1) {
    sum = sum + number;       // moved to top of loop
    System.out.print("Enter a number (-1 to quit): ");
    number = console.nextInt();
}

System.out.println("The total is " + sum);
```

Constant with sentinel

- A better solution uses a constant for the sentinel:

```
public static final int SENTINEL = -1;
```

- This solution uses the constant:

```
Scanner console = new Scanner(System.in);
int sum = 0;
System.out.print("Enter a number (" + SENTINEL + " to quit): ");
int number = console.nextInt();

while (number != SENTINEL) {
    sum = sum + number;

    System.out.print("Enter a number (" + SENTINEL + " to quit): ");
    number = console.nextInt();
}

System.out.println("The total is " + sum);
```