Building Java Programs

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

reading: 4.1, 5.1

self-check: Ch. 4 #2; Ch. 5 # 1-10

exercises: Ch. 4 #2, 4, 5, 8; Ch. 5 # 1-2

A deceptive problem...

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

  For example, the call:

  ```
  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
}
```

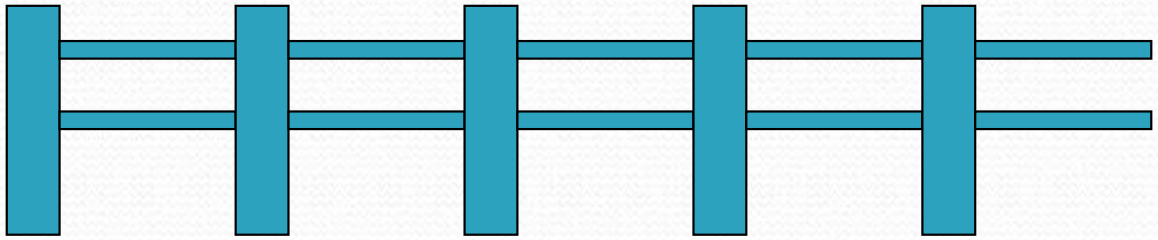
- Output from `printNumbers(5)`: `, 1, 2, 3, 4, 5`

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.
}
```

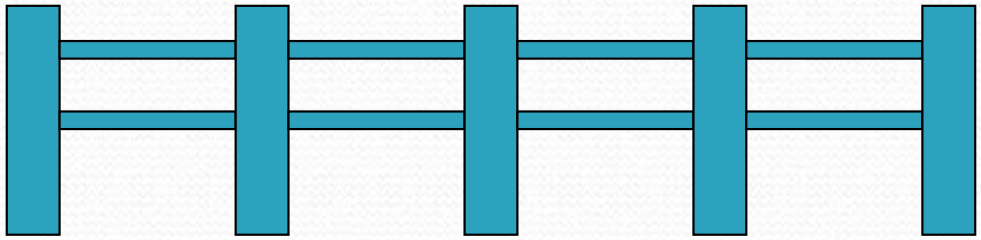


Fencepost loop

- 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

```
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:

```
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

- 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 of an integer.
  - `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 ch04-1 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

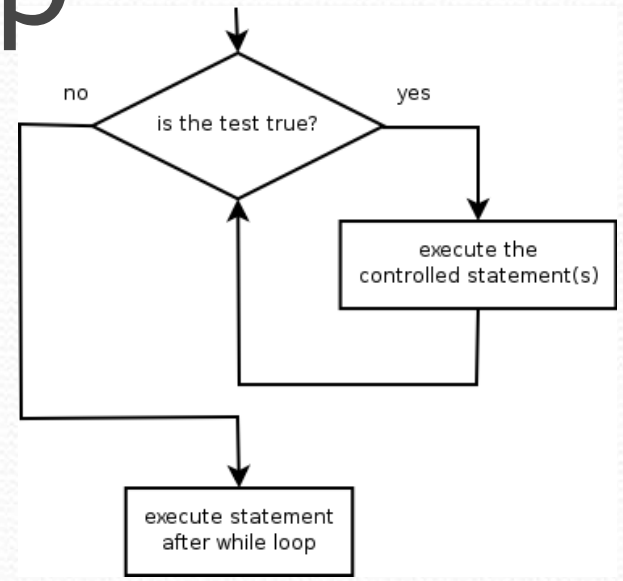
Categories of loops

- **definite loop**: Executes a known number of times.
  - The `for` loops we have seen 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 loop**: One where the number of times its body repeats is not known in advance.
  - Examples:
    - Prompt the user until they type a non-negative number.
    - Print random numbers until a prime number is printed.
    - Repeat until the user has types "q" to quit.

The `while` loop

- **`while` loop**: Repeatedly executes its body as long as a logical 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
```

Example `while` loop

```
// finds a 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` is better than `for` here because we don't know how many times we will need to increment to find the factor.

for vs. while loops

- The for loop is just a specialized form of the while loop.
  - The following loops are equivalent:

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

```
// actually, not a very compelling use of a while loop
// (a for loop is better because the # of reps is definite)
int num = 1;
while (num <= 200) {
    System.out.print(num + " ");
    num = num * 2;
}
```

while and Scanner

- while loops are often used with Scanner input.
  - You don't know many times you'll need to re-prompt the user if they type bad data. (an indefinite loop!)

- 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

```
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

videos: Ch. 5 #4

Sentinel values

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

- Example: A program that repeatedly prompts the user for numbers until the user types -1, then outputs their sum.
  - (In this case, -1 is the sentinel value.)

```
Enter a number (-1 to quit): 10
Enter a number (-1 to quit): 25
Enter a number (-1 to quit): 35
Enter a number (-1 to quit): -1
The sum is 70
```

A second sentinel problem

- Exercise: Write a program that repeatedly prompts the user for words until the user types "goodbye", then outputs the longest word that was typed.
  - (In this case, "goodbye" is the sentinel value.)

```
Type a word (or "goodbye" to quit): Obama
Type a word (or "goodbye" to quit): McCain
Type a word (or "goodbye" to quit): Biden
Type a word (or "goodbye" to quit): Palin
Type a word (or "goodbye" to quit): goodbye
The longest word you typed was "McCain" (6 letters)
```

Flawed sentinel solution

- What's wrong with this solution?

```
Scanner console = new Scanner(System.in);
String longest = "";
String word = "";   // "dummy value"; anything but "goodbye"

while (!word.equals("goodbye")) {
    System.out.print("Type a word (or \"goodbye\" to quit): ");
    word = console.next();
    if (word.length() > longest.length()) {
        longest = word;
    }
}

System.out.println("The longest word you typed was \"" +
        longest + "\" (" + longest.length() + " letters)");
```

- The solution produces the wrong output!

```
The longest word you typed was "goodbye" (7 letters)
```

The problem

- Our code uses a pattern like this:

```
longest = empty string.
while (input is not the sentinel) {
    prompt for input; read input.
    check if input is longest; if so, store it.
}
```

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

```
    prompt for input; read input ("goodbye").
    check if input is longest; if so, store it.
```

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

A fencepost solution

- We need to use a pattern like this:

```
longest = empty string.
prompt for input; read input.                    // place 1st "post"

while (input is not the sentinel) {
    check if input is longest; if so, store it.  // 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);
String longest = "";

// moved one "post" out of loop
System.out.print("Type a word (or \"goodbye\" to quit): ");
String word = console.next();

while (!word.equals("goodbye")) {
    if (word.length() > longest.length()) {
        longest = word;     // moved to top of loop
    }
    System.out.print("Type a word (or \"goodbye\" to quit): ");
    word = console.next();
}

System.out.println("The longest word you typed was \"" +
        longest + "\" (" + longest.length() + " letters)");
```

Constant with sentinel

- A better solution uses a constant for the sentinel:

```
public static final String SENTINEL = "goodbye";
```

- This solution uses the constant:

```
Scanner console = new Scanner(System.in);
System.out.print("Type a word (or \"" + SENTINEL + "\" to quit): ");
String word = console.next();
String longest = "";

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

System.out.println("The longest word you typed was \"" +
        longest + "\" (" + longest.length() + " letters)");
```

Sentinel number problem

- Solution to the "sum numbers until -1 is typed" problem:

```
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 sum is " + sum);
```

Building Java Programs

Chapter 5
Lecture 5-2: Random Numbers

reading: 5.1 - 5.2

self-check: #8 - 17

exercises: #3 - 6, 10, 12

videos: Ch. 5 #1-2

The Random class

- A Random object generates pseudo-random* numbers.
  - Class Random is found in the java.util package.

```
import java.util.*;
```

Method name	Description
nextInt()	returns a random integer
nextInt(**max**)	returns a random integer in the range [0, *max*) in other words, 0 to *max*-1 inclusive
nextDouble()	returns a random real number in the range [0.0, 1.0)

  - Example:

```
Random rand = new Random();
int randomNumber = rand.nextInt(10);   // 0-9
```

Generating random numbers

- Common usage: to get a random number from 1 to *N*

```
int n = rand.nextInt(20) + 1;   // 1-20 inclusive
```

- To get a number in arbitrary range [*min*, *max*] inclusive:

```
nextInt(size of range) + min
```

  - where (**size of range**) is (**max - min + 1**)

- Example: A random integer between 4 and 10 inclusive:

```
int n = rand.nextInt(7) + 4;
```

Random questions

- Given the following declaration, how would you get:

```
Random rand = new Random();
```

  - A random number between 1 and 100 inclusive?

```
int random1 = rand.nextInt(100) + 1;
```

  - A random number between 50 and 100 inclusive?

```
int random2 = rand.nextInt(51) + 50;
```

  - A random number between 4 and 17 inclusive?

```
int random3 = rand.nextInt(14) + 4;
```

Random and other types

- `nextDouble` method returns a `double` between 0.0 - 1.0
  - Example: Get a random GPA value between 1.5 and 4.0:

```
double randomGpa = rand.nextDouble() * 2.5 + 1.5;
```

- Any set of possible values can be mapped to integers
  - code to randomly play Rock-Paper-Scissors:

```
int r = rand.nextInt(3);
if (r == 0) {
    System.out.println("Rock");
} else if (r == 1) {
    System.out.println("Paper");
} else {
    System.out.println("Scissors");
}
```

Random question

- Write a program that simulates rolling of two 6-sided dice until their combined result comes up as 7.

```
2 + 4 = 6
3 + 5 = 8
5 + 6 = 11
1 + 1 = 2
4 + 3 = 7
You won after 5 tries!
```

  - Modify the program to play 3 dice games using a method.

Random answer

```
// Rolls two dice until a sum of 7 is reached.
import java.util.*;

public class Dice {
    public static void main(String[] args) {
        Random rand = new Random();
        int tries = 0;

        int sum = 0;
        while (sum != 7) {
            // roll the dice once
            int roll1 = rand.nextInt(6) + 1;
            int roll2 = rand.nextInt(6) + 1;
            sum = roll1 + roll2;
            System.out.println(roll1 + " + " + roll2 + " = " + sum);
            tries++;
        }

        System.out.println("You won after " + tries + " tries!");
    }
}
```

Random question

- Write a multiplication tutor program.
  - Ask user to solve problems with random numbers from 1-20.
  - The program stops after an incorrect answer.

```
14 * 8 = 112
Correct!
5 * 12 = 60
Correct!
8 * 3 = 24
Correct!
5 * 5 = 25
Correct!
20 * 14 = 280
Correct!
19 * 14 = 256
Incorrect; the answer was 266

You solved 5 correctly
```

Random answer

```
import java.util.*;

// Asks the user to do multiplication problems and scores them.
public class MultiplicationTutor {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        Random rand = new Random();

        // fencepost solution - pull first question outside of loop
        int correct = 0;
        int last = askQuestion(console, rand);
        int lastCorrect = 0;

        // loop until user gets one wrong
        while (last > 0) {
            lastCorrect = last;
            correct++;
            last = askQuestion(console, rand);
        }

        System.out.println("You solved " + correct + " correctly");
        if (correct > 0) {
            System.out.println("Last correct answer was " + lastCorrect);
        }
    }
    ...
```

Random answer 2

```
    ...

    // Asks the user one multiplication problem,
    // returning the answer if they get it right and 0 if not.
    public static int askQuestion(Scanner console, Random rand) {
        // pick two random numbers between 1 and 20 inclusive
        int num1 = rand.nextInt(20) + 1;
        int num2 = rand.nextInt(20) + 1;

        System.out.print(num1 + " * " + num2 + " = ");
        int guess = console.nextInt();
        if (guess == num1 * num2) {
            System.out.println("Correct!");
            return num1 * num2;
        } else {
            System.out.println("Incorrect; the correct answer was " +
                               (num1 * num2));
            return 0;
        }
    }
}
```

Building Java Programs

Chapter 5
Lecture 5-3: Boolean Logic

reading: 5.2

self-check: #11 - 17

exercises: #12

videos: Ch. 5 #2

`while` loop question

- Write a method named `digitSum` that accepts an integer as a parameter and returns the sum of the digits of that number.
  - `digitSum(29107)` returns 2+9+1+0+7 or `19`
  - Assume that the number is non-negative.
  - Hint: Use the `%` operator to extract a digit from a number.

`while` loop answer

- The following code implements the method:

```
public static int digitSum(int n) {
    int sum = 0;
    while (n > 0) {
        sum = sum + (n % 10); // add last digit to sum
        n = n / 10;           // remove last digit
    }
    return sum;
}
```

Type `boolean`

- **`boolean`**: A logical type whose values are `true` and `false`.
  - A **test** in an `if`, `for`, or `while` is a `boolean` expression.
  - You can create `boolean` variables, pass `boolean` parameters, return `boolean` values from methods, ...

```
boolean minor = (age < 21);
boolean expensive = iPhonePrice > 200.00;
boolean iLoveCS = true;

if (minor) {
    System.out.println("Can't purchase alcohol!");
}
if (iLoveCS || !expensive) {
    System.out.println("Buying an iPhone");
}
```

Methods that return `boolean`

- Methods can return `boolean` values.
  - A call to such a method can be a loop or `if` **test**.

```
Scanner console = new Scanner(System.in);
System.out.print("Type your name: ");
String line = console.nextLine();

if (line.startsWith("Dr.")) {
    System.out.println("Will you marry me?");
} else if (line.endsWith(", Esq.")) {
    System.out.println("And I am Ted 'Theodore' Logan!");
}
```

De Morgan's Law

- **De Morgan's Law**:
  Rules used to *negate* or *reverse* boolean expressions.
  - Useful when you want the opposite of a known boolean test.

Original Expression	Negated Expression	Alternative
`a && b`	`!a || !b`	`!(a && b)`
`a || b`	`!a && !b`	`!(a || b)`

  - Example:

Original Code	Negated Code

`if (x == 7 && y > 3) {`
`  ...`
`}`
	`if (x != 7 || y <= 3) {`
	`  ...`
	`}`

Writing `boolean` methods

```
public static boolean bothOdd(int n1, int n2) {
    if (n1 % 2 != 0 && n2 % 2 != 0) {
        return true;
    } else {
        return false;
    }
}
```

- Calls to this methods can now be used as tests:

```
if (bothOdd(7, 13)) {
    ...
}
```

"Boolean Zen", part 1

- Students new to `boolean` often test if a result is `true`:

```
if (bothOdd(7, 13) == true) {     // bad
    ...
}
```

- But this is unnecessary and redundant. Preferred:

```
if (bothOdd(7, 13)) {             // good
    ...
}
```

- A similar pattern can be used for a `false` test:

```
if (bothOdd(7, 13) == false) {    // bad
if (!bothOdd(7, 13)) {            // good
```

"Boolean Zen", part 2

- Methods that return `boolean` often have an `if/else` that returns `true` or `false`:

```
public static boolean bothOdd(int n1, int n2) {
    if (n1 % 2 != 0 && n2 % 2 != 0) {
        return true;
    } else {
        return false;
    }
}
```

  - But the code above is unnecessarily verbose.

Solution w/ `boolean` variable

- We could store the result of the logical test.

```
public static boolean bothOdd(int n1, int n2) {
    boolean test = (n1 % 2 != 0 && n2 % 2 != 0);
    if (test) {   // test == true
        return true;
    } else {      // test == false
        return false;
    }
}
```

  - Notice: Whatever `test` is, we want to return that.
    - If `test` is `true` , we want to return `true`.
    - If `test` is `false`, we want to return `false`.

Solution w/ "Boolean Zen"

- Observation: The `if/else` is unnecessary.
  - The variable `test` stores a `boolean` value;
    its value is exactly what you want to return. So return that!

```
public static boolean bothOdd(int n1, int n2) {
    boolean test = (n1 % 2 != 0 && n2 % 2 != 0);
    return test;
}
```

- An even shorter version:
  - We don't even need the variable `test`.
    We can just perform the test and return its result in one step.

```
public static boolean bothOdd(int n1, int n2) {
    return (n1 % 2 != 0 && n2 % 2 != 0);
}
```

"Boolean Zen" template

- Replace

```
public static boolean name(parameters) {
    if (test) {
        return true;
    } else {
        return false;
    }
}
```

- with

```
public static boolean name(parameters) {
    return test;
}
```

Boolean question

- Write a program that prompts the user for two words and reports whether they "rhyme" (end with the same last two letters) and/or "alliterate" (start with the same letter).

(run #1)

```
Type two words: car STAR
They rhyme!
```

(run #2)

```
Type two words: Bare blare
They rhyme!
They alliterate!
```

(run #3)

```
Type two words: booyah socks
They have nothing in common.
```

Boolean answer

```
public static void main(String[] args) {
    Scanner console = new Scanner(System.in);
    System.out.print("Type two words: ");
    String word1 = console.next();          // Type two words: car STAR
    String word2 = console.next();          // They rhyme!

    if (rhyme(word1, word2)) {
        System.out.println("They rhyme!");
    }
    if (alliterate(word1, word2)) {
        System.out.println("They alliterate (start with the same letter)!");
    }
}

// Returns true if s1 and s2 end with the same two letters.
public static boolean rhyme(String s1, String s2) {
    return s2.length() >= 2 && s1.endsWith(s2.substring(s2.length() - 2));
}

// Returns true if s1 and s2 start with the same letter.
public static boolean alliterate(String s1, String s2) {
    return s1.startsWith(s2.substring(0, 1));
}
```

Boolean practice questions

- Write a method named `isVowel` that returns whether a `String` is a vowel (a, e, i, o, or u), case-insensitively.
  - `isVowel("q")` returns `false`
  - `isVowel("A")` returns `true`
  - `isVowel("e")` returns `true`

- Change the above method into an `isNonVowel` that returns whether a `String` is any character EXCEPT a vowel (a, e, i, o, or u).
  - `isNonVowel("q")` returns `true`
  - `isNonVowel("A")` returns `false`
  - `isNonVowel("e")` returns `false`

- Write methods named `allVowels` and `containsVowel`.

Boolean practice answers

```
public static boolean isVowel(String s) {
    if (s.equalsIgnoreCase("a") || s.equalsIgnoreCase("e") ||
        s.equalsIgnoreCase("i") || s.equalsIgnoreCase("o") ||
        s.equalsIgnoreCase("u")) {
        return true;
    } else {
        return false;
    }
}

public static boolean isNonVowel(String s) {
    if (!s.equalsIgnoreCase("a") && !s.equalsIgnoreCase("e") &&
        !s.equalsIgnoreCase("i") && !s.equalsIgnoreCase("o") &&
        !s.equalsIgnoreCase("u")) {
        return true;
    } else {
        return false;
    }
}
```

Boolean practice answers 2

```
// Enlightened version.  I have seen the true way (and false way)
public static boolean isVowel(String s) {
    return s.equalsIgnoreCase("a") || s.equalsIgnoreCase("e") ||
           s.equalsIgnoreCase("i") || s.equalsIgnoreCase("o") ||
           s.equalsIgnoreCase("u");
}
```

```
// Enlightened version
public static boolean isNonVowel(String s) {
    return !s.equalsIgnoreCase("a") && !s.equalsIgnoreCase("e") &&
           !s.equalsIgnoreCase("i") && !s.equalsIgnoreCase("o") &&
           !s.equalsIgnoreCase("u");
}
```

When to return?

- In methods that involve a loop and a `boolean` return:
  - How do you figure out whether to return `true` or `false`?
  - When should the method return its result?

- Example problem:
  - Write a method `seven` that accepts a `Random` parameter and uses it to pick up to 10 lotto numbers between 1 and 30.
  - The method should print each number as it is drawn.
    - Example output from 2 calls:

```
15 29 18 29 11 3 30 17 19 22
29 5 29 16 4 7
```

  - If any of the numbers is a lucky 7, the method should return `true`. Otherwise, it should return `false`.

Flawed solution

- Common incorrect solution:

```
// Draws 10 random lotto numbers.
// Returns true if one of them is a lucky 7.
public static boolean seven(Random rand) {
    for (int i = 1; i <= 10; i++) {
        int num = rand.nextInt(30) + 1;
        System.out.print(num + " ");
        if (num == 7) {
            return true;
        } else {
            return false;
        }
    }
}
```

  - The method tries to return immediately after the first roll.
  - This is bad, if that roll isn't a 7; we need to roll all 10 times to see if any of them is a 7.

Returning at the right time

- Corrected code:

```
// Draws 10 random lotto numbers.
// Returns true if one of them is a lucky 7.
public static boolean seven(Random rand) {
    for (int i = 1; i <= 10; i++) {
        int num = rand.nextInt(30) + 1;
        System.out.print(num + " ");
        if (num == 7) {   // found lucky 7; can exit now
            return true;
        }
    }

    // if we get here, we know there was no 7
    return false;
}
```

  - Returns immediately if 7 is found, because the answer must be `true`. If 7 isn't found, we draw the next lotto number. If all 10 aren't 7, the loop ends and we return `false`.

Boolean return questions

- Write a method named `hasAnOddDigit` that returns whether any digit of a positive integer is odd.
  - `hasAnOddDigit(4822116)` returns `true`
  - `hasAnOddDigit(2448)` returns `false`

- Write a method named `allDigitsOdd` that returns whether every digit of a positive integer is odd.
  - `allDigitsOdd(135319)` returns `true`
  - `allDigitsOdd(9175293)` returns `false`

- Write a method named `isAllVowels` that returns `true` if every character in a `String` is a vowel, else `false`.
  - `isAllVowels("eIeIo")` returns `true`
  - `isAllVowels("oink")` returns `false`

Boolean return answers

```
public static boolean hasAnOddDigit(int n) {
    while (n > 0) {
        if (n % 2 != 0) {    // check whether last digit is odd
            return true;
        }
        n = n / 10;
    }
    return false;
}

public static boolean allDigitsOdd(int n) {
    while (n > 0) {
        if (n % 2 == 0) {    // check whether last digit is even
            return false;
        }
        n = n / 10;
    }
    return true;
}

public static boolean isAllVowels(String s) {
    for (int i = 0; i < s.length(); i++) {
        String letter = s.substring(i, i + 1);
        if (!isVowel(letter)) {
            return false;
        }
    }
    return true;
}
```

Building Java Programs

Chapter 5

Lecture 5-3: Assertions, `do/while` loops

reading: 5.4 - 5.5

self-check: 22-24, 26-28

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
```

- 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.
  - Often around 1/2 of the correct answers are "sometimes."

Assertion example 1

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

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

        // Point C
        z++;

        // 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	SOMETIMES	SOMETIMES
Point D	SOMETIMES	SOMETIMES	NEVER
Point E	ALWAYS	NEVER	SOMETIMES

Assertion example 2

```
public static int mystery(Scanner console) {
    int prev = 0;
    int count = 0;
    int next = console.nextInt();
    // Point A
    while (next != 0) {
        // Point B
        if (next == prev) {
            // Point C
            count++;
        }
        prev = next;
        next = console.nextInt();
        // Point D
    }
    // Point E
    return count;
}
```

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

	next == 0	prev == 0	next == prev
Point A	SOMETIMES	ALWAYS	SOMETIMES
Point B	NEVER	SOMETIMES	SOMETIMES
Point C	NEVER	NEVER	ALWAYS
Point D	SOMETIMES	NEVER	SOMETIMES
Point E	ALWAYS	SOMETIMES	SOMETIMES

Assertion example 3

```
// Assumes y >= 0, and returns x^y
public static int pow(int x, int y) {
    int prod = 1;

    // Point A
    while (y > 0) {
        // Point B
        if (y % 2 == 0) {
            // Point C
            x = x * x;
            y = y / 2;
            // Point D
        } else {
            // Point E
            prod = prod * x;
            y--;
            // Point F
        }
    }
    // Point G
    return prod;
}
```

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

	y > 0	y % 2 == 0
Point A	SOMETIMES	SOMETIMES
Point B	ALWAYS	SOMETIMES
Point C	ALWAYS	ALWAYS
Point D	ALWAYS	SOMETIMES
Point E	ALWAYS	NEVER
Point F	SOMETIMES	ALWAYS
Point G	NEVER	ALWAYS

`while` loop variations

reading: 5.4

self-checks: #22-24

exercises: #6

The do/while loop

- **do/while loop**: Executes statements repeatedly while a condition is `true`, testing it at the *end* of each repetition.

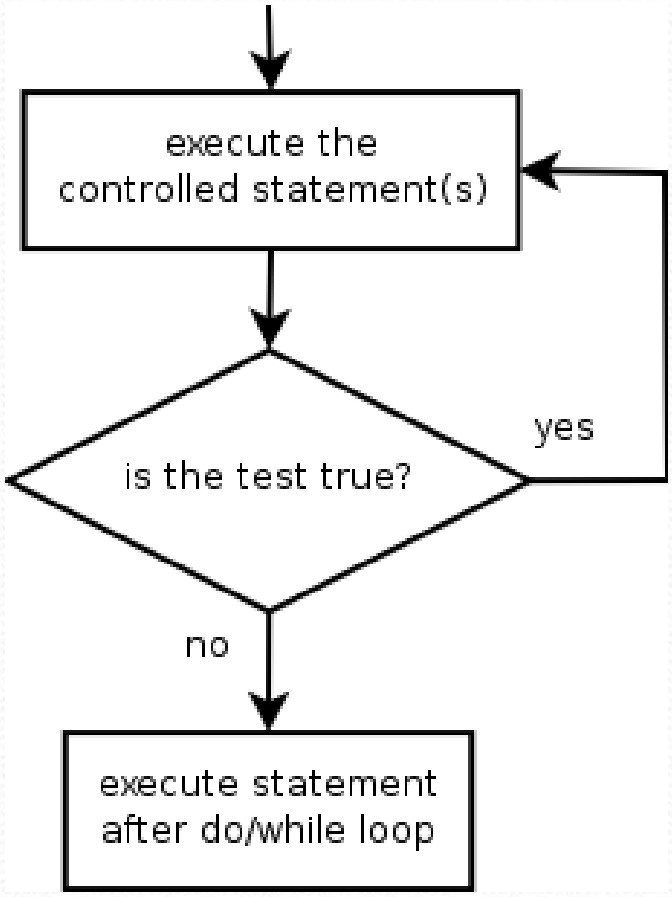
```
do {
    statement(s);
} while (test);
```

  - Example:

```
// prompt until the user gets the right password
String phrase;
do {
    System.out.print("Password: ");
    phrase = console.next();
} while (!phrase.equals("abracadabra"));
```

do/while flow chart

- How does this differ from the while loop?
  - The controlled **statement(s)** will always execute the first time, regardless of whether the **test** is `true` or `false`.



do/while question

- Modify the previous Dice program to use do/while.
  - Example log of execution:

```
2 + 4 = 6
3 + 5 = 8
5 + 6 = 11
1 + 1 = 2
4 + 3 = 7
You won after 5 tries!
```

- Modify the previous Sentinel program to use do/while.
  - Is do/while a good fit for solving this problem?

do/while answer

```
// Rolls two dice until a sum of 7 is reached.
import java.util.*;

public class Dice {
    public static void main(String[] args) {
        Random rand = new Random();
        int tries = 0;
        int sum;
        do {
            int roll1 = rand.nextInt(6) + 1;
            int roll2 = rand.nextInt(6) + 1;
            sum = roll1 + roll2;
            System.out.println(roll1 + " + " + roll2 + " = " + sum);
            tries++;
        } while (sum != 7);

        System.out.println("You won after " + tries + " tries!");
    }
}
```

break

- **`break`** statement: Immediately exits a loop.
  - Can be used to write a loop whose test is in the middle.
  - Such loops are often called *"forever" loops* because their header's boolean test is often changed to a trivial `true`.

```
while (true) {
    statement(s);
    if (test) {
        break;
    }
    statement(s);
}
```

  - `break` is bad style! Do not use it on CSE 142 homework.

Sentinel loop with break

- A working sentinel loop solution using break:

```
Scanner console = new Scanner(System.in);
int sum = 0;
while (true) {
    System.out.print("Enter a number (-1 to quit): ");
    int number = console.nextInt();
    if (number == -1) {     // don't add -1 to sum
        break;
    }
    sum = sum + number;     // number != -1 here
}

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