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:
  ```plaintext
  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.

```c
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.

```plaintext
place a post.
for (length of fence - 1) {
  place some wire.
  place a post.
}
```
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
}
• 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.
// 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
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 types "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
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 numbers until the user types 0, then outputs their sum.
  - (In this case, 0 is the sentinel value.)

  Enter a number (0 to quit): 10
  Enter a number (0 to quit): 20
  Enter a number (0 to quit): 30
  Enter a number (0 to quit): 0
  The sum is 60
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);
• Modify your program to use a sentinel value of -1.
  – Example log of execution:

  Enter a number (-1 to quit): 15
  Enter a number (-1 to quit): 25
  Enter a number (-1 to quit): 10
  Enter a number (-1 to quit): 30
  Enter a number (-1 to quit): -1
  The total is 80
• To see the problem, change the sentinel's value to -1:

```java
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 = sum + number;
}

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

• Now the solution produces the wrong output. Why?
The total was 79
• 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.
  – Must read \(N\) numbers, but only sum the first \(N-1\) of them.
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" style solution by pulling some code out of the loop.
Scanner console = new Scanner(System.in);
int sum = 0;

// pull one prompt/read ("post") out of the loop
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);
public static final int SENTINEL = -1;

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

// pull one prompt/read ("post") out of the loop
System.out.print("Enter a number (" + SENTINEL + " to quit): ");
int number = console.nextInt();

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

System.out.println("The total is " + sum);
Random numbers
A Random object generates pseudo-random numbers.

- Class Random is found in the java.util package.
  
  ```java
  import java.util.*;
  ```

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nextInt()</code></td>
<td>returns a random integer</td>
</tr>
<tr>
<td><code>nextInt(max)</code></td>
<td>returns a random integer in the range ([0, max)) in other words, 0 to (max-1) inclusive</td>
</tr>
<tr>
<td><code>nextDouble()</code></td>
<td>returns a random real number in the range ([0.0, 1.0))</td>
</tr>
</tbody>
</table>

**Example:**

```java
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

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

• To get a number in arbitrary range \([min, max]\) inclusive:

```java
name.nextInt(size of range) + min
```
  • where \((size of range)\) is \((max - min + 1)\)

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

```java
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 47 inclusive?
    int random1 = rand.nextInt(47) + 1;

  – A random number between 23 and 30 inclusive?
    int random2 = rand.nextInt(8) + 23;

  – A random even number between 4 and 12 inclusive?
    int random3 = rand.nextInt(5) * 2 + 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:
    ```java
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:
    ```java
    int r = rand.nextInt(3);
    if (r == 0) {
        System.out.println("Rock");
    } else if (r == 1) {
        System.out.println("Paper");
    } else {  // r == 2
        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!
// 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!");
    }
}
• Write a program that plays an adding game.
  – Ask user to solve random adding problems with 2-5 numbers.
  – The user gets 1 point for a correct answer, 0 for incorrect.
  – The program stops after 3 incorrect answers.

\[
4 + 10 + 3 + 10 = \underline{27}
\]
\[
9 + 2 = \underline{11}
\]
\[
8 + 6 + 7 + 9 = \underline{25}
\]
Wrong! The answer was 30
\[
5 + 9 = \underline{13}
\]
Wrong! The answer was 14
\[
4 + 9 + 9 = \underline{22}
\]
\[
3 + 1 + 7 + 2 = \underline{13}
\]
\[
4 + 2 + 10 + 9 + 7 = \underline{42}
\]
Wrong! The answer was 32
You earned 4 total points.
// Asks the user to do adding problems and scores them.
import java.util.*;

public class AddingGame {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);
        Random rand = new Random();

        // play until user gets 3 wrong
        int points = 0;
        int wrong = 0;
        while (wrong < 3) {
            int result = play(console, rand);  // play one game
            if (result > 0) {
                points++;
            } else {
                wrong++;
            }
        }

        System.out.println("You earned " + points + " total points.");
    }
}
... // Builds one addition problem and presents it to the user. // Returns 1 point if you get it right, 0 if wrong.
public static int play(Scanner console, Random rand) {
    // print the operands being added, and sum them
    int operands = rand.nextInt(4) + 2;
    int sum = rand.nextInt(10) + 1;
    System.out.print(sum);
    for (int i = 2; i <= operands; i++) {
        int n = rand.nextInt(10) + 1;
        sum += n;
        System.out.print(" + " + n);
    }
    System.out.print(" = ");

    // read user's guess and report whether it was correct
    int guess = console.nextInt();
    if (guess == sum) {
        return 1;
    } else {
        System.out.println("Wrong! The answer was " + total);
        return 0;
    }
}
The **do/while loop**

- **do/while loop**: Performs its test at the *end* of each repetition.
  - Guarantees that the loop's `{}` body will run at least once.

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

// Example: prompt until correct password is typed
String phrase;
do {
    System.out.print("Type your password: ");
    phrase = console.next();
} while (!phrase.equals("abracadabra"));
do/while question

- Modify the previous Dice program to use do/while.

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

- Is do/while a good fit for our past Sentinel program?
// 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;  // one roll
            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!");
    }
}
Type boolean
Some methods return logical values. A call to such a method is used as a test in a loop or if.

```java
Scanner console = new Scanner(System.in);
System.out.print("Type your first name: ");
String name = console.next();

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>equals(&lt;str&gt;)</td>
<td>whether two strings contain the same characters</td>
</tr>
<tr>
<td>equalsIgnoreCase(&lt;str&gt;)</td>
<td>whether two strings contain the same characters, ignoring upper vs. lower case</td>
</tr>
<tr>
<td>startsWith(&lt;str&gt;)</td>
<td>whether one contains other's characters at start</td>
</tr>
<tr>
<td>endsWith(&lt;str&gt;)</td>
<td>whether one contains other's characters at end</td>
</tr>
<tr>
<td>contains(&lt;str&gt;)</td>
<td>whether the given string is found within this one</td>
</tr>
</tbody>
</table>

```java
String name = console.next();
if (name.contains("Prof")) {
    System.out.println("When are your office hours?");
} else if (name.equalsIgnoreCase("STUART")) {
    System.out.println("Let's talk about meta!");
}
```
**Type boolean**

- **boolean**: A logical type whose values are **true** and **false**.
  - A logical **test** is actually a **boolean** expression.
  - It is legal to:
    - create a **boolean** variable
    - pass a **boolean** value as a parameter
    - return a **boolean** value from methods
    - call a method that returns a **boolean** and use it as a test

```java
boolean minor    = (age < 21);
boolean isProf   = name.contains("Prof");
boolean lovesCSE = true;

// allow only CSE-loving students over 21
if (minor || isProf || !lovesCSE) {
    System.out.println("Can't enter the club!");
}
```
Using `boolean`

- **Why is type `boolean` useful?**
  - Can capture a complex logical test result and use it later
  - Can write a method that does a complex test and returns it
  - Makes code more readable
  - Can pass around the result of a logical test (as param/return)

```java
boolean goodAge = age >= 12 && age < 29;
boolean goodHeight = height >= 78 && height < 84;
boolean rich = salary >= 100000.0;

if ((goodAge && goodHeight) || rich) {
    System.out.println("Okay, let's go out!");
} else {
    System.out.println("It's not you, it's me...");
}
```
public static boolean isPrime(int n) {
    int factors = 0;
    for (int i = 1; i <= n; i++) {
        if (n % i == 0) {
            factors++;
        }
    }
    if (factors == 2) {
        return true;
    } else {
        return false;
    }
}

• Calls to methods returning boolean can be used as tests:
  if (isPrime(57)) {
      ...
  }

Boolean question

• Improve our "rhyme" / "alliterate" program to use boolean methods to test for rhyming and alliteration.

Type two words: Bare blare
They rhyme!
They alliterate!
if (rhyme(word1, word2)) {
    System.out.println("They rhyme!");
} 
if (alliterate(word1, word2)) {
    System.out.println("They alliterate!");
}
...

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

// Returns true if s1 and s2 start with the same letter.
public static boolean alliterate(String s1, String s2) {
    if (s1.startsWith(s2.substring(0, 1))) {
        return true;
    } else {
        return false;
    }
}
• Students new to boolean often test if a result is true:

```java
if (isPrime(57) == true) {   // bad
    ...
}
```

• But this is unnecessary and redundant. Preferred:

```java
if (isPrime(57)) {   // good
    ...
}
```

• A similar pattern can be used for a false test:

```java
if (isPrime(57) == false) {   // bad
    ...
}
if (!isPrime(57)) {   // good
```
• Methods that return boolean often have an if/else that returns true or false:

```java
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.
• We could store the result of the logical test.

```java
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!

```java
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.

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

• Replace

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

• with

```java
public static boolean name(parameters) {
    return test;
}
```
**Improved isPrime method**

- The following version utilizes Boolean Zen:

  ```java
  public static boolean isPrime(int n) {
      int factors = 0;
      for (int i = 1; i <= n; i++) {
          if (n % i == 0) {
              factors++;
          }
      }
      return factors == 2;  // if n has 2 factors, true
  }
  ```

- Modify our Rhyme program to use Boolean Zen.
public static void main(String[] args) {
    Scanner console = new Scanner(System.in);
    System.out.print("Type two words: ");
    String word1 = console.next().toLowerCase();
    String word2 = console.next().toLowerCase();

    if (rhyme(word1, word2)) {
        System.out.println("They rhyme!");
    }
    if (alliterate(word1, word2)) {
        System.out.println("They alliterate!");
    }
}

// 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));
}
"Short-circuit" evaluation

- Java stops evaluating a test if it knows the answer.
  - `&&` stops early if any part of the test is `false`.
  - `||` stops early if any part of the test is `true`.

- The following test will crash if s2's length is less than 2:

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

- The following test will not crash; it stops if length < 2:

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

- **De Morgan's Law:** Rules used to negate boolean tests.
  - Useful when you want the opposite of an existing test.

<table>
<thead>
<tr>
<th>Original Expression</th>
<th>Negated Expression</th>
<th>Alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a &amp;&amp; b</code></td>
<td>`!a</td>
<td></td>
</tr>
<tr>
<td>`a</td>
<td></td>
<td>b`</td>
</tr>
</tbody>
</table>

- Example:

<table>
<thead>
<tr>
<th>Original Code</th>
<th>Negated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if (x == 7 &amp;&amp; y &gt; 3) {</code></td>
<td>`if (x != 7</td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>}</code></td>
</tr>
</tbody>
</table>
• 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.
  - `isNonVowel("q")` returns `true`
  - `isNonVowel("A")` returns `false`
  - `isNonVowel("e")` returns `false`
// 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 "Boolean Zen" version
public static boolean isNonVowel(String s) {
    return !s.equalsIgnoreCase("a") && !s.equalsIgnoreCase("e") &&
    !s.equalsIgnoreCase("i") && !s.equalsIgnoreCase("o") &&
    !s.equalsIgnoreCase("u");

    // or, return !isVowel(s);
}
When to return?

• Methods with loops and return values can be tricky.
  – When and where should the method return its result?

• Write a method `seven` that accepts a `Random` parameter and uses it to draw up to ten lotto numbers from 1-30.
  – If any of the numbers is a lucky 7, the method should stop and return `true`. If none of the ten are 7 it should return `false`.
  – The method should print each number as it is drawn.

  15 29 18 29 11 3 30 17 19 22  (first call)
  29 5 29 4 7  (second call)
// Draws 10 lotto numbers; returns true if one is 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 always returns immediately after the first roll.
- This is wrong if that roll isn't a 7; we need to keep rolling.
Returning at the right time

// Draws 10 lotto numbers; returns true if one is 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;
        }
    }
    return false;  // if we get here, there was no 7
}

- Returns true immediately if 7 is found.
- If 7 isn't found, the loop continues drawing lotto numbers.
- If all ten aren't 7, the loop ends and we return false.
• Write a method `digitSum` that accepts an integer parameter and returns the sum of its digits.
  
  – Assume that the number is non-negative.
  
  – Example: `digitSum(29107)` returns \(2+9+1+0+7\) or 19

  – Hint: Use the \(\%\) operator to extract a digit from a number.
public static int digitSum(int n) {
    n = Math.abs(n); // handle negatives
    int sum = 0;
    while (n > 0) {
        sum = sum + (n % 10); // add last digit
        n = n / 10; // remove last digit
    }
    return sum;
}
Boolean return questions

- **hasAnOddDigit**: returns `true` if **any** digit of an integer is odd.
  - `hasAnOddDigit(4822116)` returns `true`
  - `hasAnOddDigit(2448)` returns `false`

- **allDigitsOdd**: returns `true` if **every** digit of an integer is odd.
  - `allDigitsOdd(135319)` returns `true`
  - `allDigitsOdd(9174529)` returns `false`

- **isAllVowels**: returns `true` if **every** char in a String is a vowel.
  - `isAllVowels("eIeIo")` returns `true`
  - `isAllVowels("oink")` returns `false`

- These problems are available in our Practice-It! system under **5.x**.
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;
}
Logical Assertions
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.
• Suppose you have the following code:

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

```java
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:
  \[
  \text{int } x = 3; \\
  // \text{ is } x > 0? \text{ ALWAYS}
  \]

• In general you know nothing about parameters' values:
  \[
  \text{public static void mystery(int a, int b) } \{ \\
  // \text{ is } a == 10? \text{ SOMETIMES}
  \}
  \]

• But inside an if, while, etc., you may know something:
  \[
  \text{public static void mystery(int a, int b) } \{ \\
  \text{if (a < 0) } \{ \\
  // \text{ is } a == 10? \text{ NEVER}
  \text{...}
  \}
  \}
  \]
Assertions and loops

• At the start of a loop's body, the loop's test must be true:
  ```java
  while (y < 10) {
    // is y < 10?  ALWAYS
    ...
  }
  ```

• After a loop, the loop's test must be false:
  ```java
  while (y < 10) {
    ...
  }
  // is y < 10?  NEVER
  ```

• Inside a loop's body, the loop's test may become false:
  ```java
  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.
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.

<table>
<thead>
<tr>
<th></th>
<th>x &lt; y</th>
<th>x == y</th>
<th>z == 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point A</td>
<td>SOMETIMES</td>
<td>SOMETIMES</td>
<td>ALWAYS</td>
</tr>
<tr>
<td>Point B</td>
<td>NEVER</td>
<td>SOMETIMES</td>
<td>SOMETIMES</td>
</tr>
<tr>
<td>Point C</td>
<td>SOMETIMES</td>
<td>NEVER</td>
<td>NEVER</td>
</tr>
<tr>
<td>Point D</td>
<td>SOMETIMES</td>
<td>SOMETIMES</td>
<td>NEVER</td>
</tr>
<tr>
<td>Point E</td>
<td>ALWAYS</td>
<td>NEVER</td>
<td>SOMETIMES</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th></th>
<th>next == 0</th>
<th>prev == 0</th>
<th>next == prev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point A</td>
<td>SOMETIMES</td>
<td>ALWAYS</td>
<td>SOMETIMES</td>
</tr>
<tr>
<td>Point B</td>
<td>NEVER</td>
<td>SOMETIMES</td>
<td>SOMETIMES</td>
</tr>
<tr>
<td>Point C</td>
<td>NEVER</td>
<td>NEVER</td>
<td>ALWAYS</td>
</tr>
<tr>
<td>Point D</td>
<td>SOMETIMES</td>
<td>NEVER</td>
<td>SOMETIMES</td>
</tr>
<tr>
<td>Point E</td>
<td>ALWAYS</td>
<td>SOMETIMES</td>
<td>SOMETIMES</td>
</tr>
</tbody>
</table>
// 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;
        } else {
            // Point E
            prod = prod * x;
            y--;
        }
        // Point D
    } else {
        // Point E
        prod = prod * x;
        y--;
    }
    // Point F
}

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

<table>
<thead>
<tr>
<th></th>
<th>y &gt; 0</th>
<th>y % 2 == 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Point A</td>
<td>SOMETHING</td>
<td>SOMETHING</td>
</tr>
<tr>
<td>Point B</td>
<td>ALWAYS</td>
<td>SOMETHING</td>
</tr>
<tr>
<td>Point C</td>
<td>ALWAYS</td>
<td>ALWAYS</td>
</tr>
<tr>
<td>Point D</td>
<td>ALWAYS</td>
<td>SOMETHING</td>
</tr>
<tr>
<td>Point E</td>
<td>ALWAYS</td>
<td>NEVER</td>
</tr>
<tr>
<td>Point F</td>
<td>SOMETHING</td>
<td>ALWAYS</td>
</tr>
<tr>
<td>Point G</td>
<td>NEVER</td>
<td>ALWAYS</td>
</tr>
</tbody>
</table>