Text processing

Readings: 4.4 (pg. 235 – 237)

Characters

- **`char`**: A *primitive* type representing single characters.
- Individual characters inside a `String` are stored as `char` values.
- Literal `char` values are surrounded with apostrophe (single-quote) marks, such as `'a'` or `'4'` or `'\n'` or `'\''`
- Like any other type, you can create variables, parameters, and returns of type `char`.

```
char letter = 'S';
System.out.println(letter);    // S
```


Fun with `char`!

- `char` values can be concatenated with strings.

```
char initial = 'P';
System.out.println(initial + ". Diddy");
```

- You can compare `char` values with relational operators.
  - `'a' < 'b'` and `'Q' != 'q'`
  - *Caution*: You cannot use these operators on a `String`!
- Example:

```
// print the alphabet
for (char c = 'a'; c <= 'z'; c++) {
      System.out.print(c);
}
```


The `charAt` method

- The characters of a string can be accessed using the `String` object's `charAt` method.
  - Recall that string indices start at 0.

```
String word = console.next();
char firstLetter = word.charAt(0);
if (firstLetter == 'c') {
    System.out.println("C is for cookie!");
}
```


`char` vs. `String`

- `'h'` is a `char`

```
char c = 'h';
```

  - `char` values are **primitive**; you cannot call methods on them
  - can't say `c.length()` or `c.toUpperCase()`
- `"h"` is a `String`

```
String s = "h";
```

  - Strings are objects; they contain methods that can be called
  - *can* say `s.length()` `1`
  - *can* say `s.toUpperCase()` `"H"`
  - *can* say `s.charAt(0)` `'h'`

Text processing

- **text processing**: Examining, editing, formatting text.
- Text processing often involves `for` loops that examine the characters of a string one by one.
- You can use `charAt` to search for or count occurrences of a particular character in a string.

Text processing: Example

```
// Returns the count of occurrences of c in s.
public static int count(String s, char c) {
    int count = 0;
    for (int i = 0; i < s.length(); i++) {
        if (s.charAt(i) == c) {
            count++;
        }
    }
    return count;
}
```

- `count("mississippi", 'i')` returns 4

`String`s and `char`s: Exercises

- Recall the `String` methods

Method name	Description
`charAt(`*index*`)`	returns the character at the given index
`indexOf(`*str*`)`	returns the index where the start of the given string appears in this string (-1 if not found)
`length()`	returns the number of characters in this string
`substring(`*index1*`,`*index2*`)`	returns the characters in this string from *index1* up to, but not including, *index2*
`toLowerCase()`	returns a new string with all lowercase letters
`toUpperCase()`	returns a new string with all uppercase letters

`String`s and `char`s: Exercises

- Write a method named `pigLatinWord` that accepts a `String` as a parameter and outputs that word in simplified Pig Latin, by placing the word's first letter at the end followed by the suffix *ay*.
  - `pigLatinWord("hello")` prints `ello-hay`
  - `pigLatinWord("goodbye")` prints `oodbye-gay`

- Solution:

```
public static void pigLatinWord(String word) {
    String pigLatin = word.substring(1, word.length()) +
                      "-" + word.charAt(0) + "ay";
    System.out.println(pigLatin);
}
```


`String`s and `char`s: Exercises

- Write a method `printName` that accepts a full name as a parameter, and prints the last name followed by a comma, followed by the first name and middle initial.

```
printName("Walker Texas Ranger");
```

would output:

```
Ranger, Walker T.
```


`printName`: One possible solution

```
public static void printName(String fullName) {
    int firstBlankIndex = fullName.indexOf(" ");
    String upToMiddleInitial = fullName.substring(0, firstBlankIndex + 2);

    String middleAndLastName = fullName.substring(firstBlankIndex + 1,
                                                  fullName.length());
    int secondBlankIndex = middleAndLastName.indexOf(" ");

    // Notice that "secondBlankIndex" is used with "middleAndLastName" and NOT
    // "fullName".  If you said
    //
    // fullName.substring(secondBlankIndex + 1, fullName.length())
    //
    // you wouldn't get the last name properly.  Make sure you understand
    // why.
    String lastName = middleAndLastName.substring(secondBlankIndex + 1,
                                                  middleAndLastName.length());

    System.out.println(lastName + ", " + upToMiddleInitial + ".");
}
```


More text processing: Comparing strings

- Objects (such as `String`, `Point`, and `Color`) should be compared for equality by calling a method named `equals`.

- Example:

```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
if (name.equals("Barney")) {
    System.out.println("I love you, you love me,");
    System.out.println("We're a happy family!");
}
```

What happens if you use ==?

- Relational operators such as < and == only behave correctly on primitive values.
  - The == operator on `String`s often evaluates to `false` even when the `String`s have the same letters in them.

Example: **WRONG!**

```
String name = console.next();
if (name == "Barney") {
    System.out.println("I love you, you love me,");
    System.out.println("We're a happy family!");
}
```

- This example code will compile, but it will never print the message, even if the user does type `Barney`.

Optional: Why objects use equals vs. ==

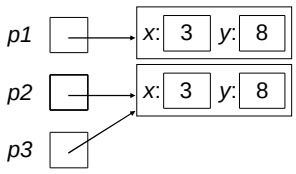
- The == operator compares whether two variables contain the same value.
- Question: What do object variables contain?
- Answer: Object variables contain addresses.
  - Using == checks if two object variables have the same address (i.e. that they refer to the same object).

Optional: Why objects use equals vs. ==

- The `equals` method compares whether two objects have the same state as each other.
- What does the following print?

```
Point p1 = new Point(3, 8);
Point p2 = new Point(3, 8);
Point p3 = p2;

if (p1 == p2) {
    System.out.println("1");
}
if (p1.equals(p2)) {
    System.out.println("2");
}
if (p2 == p3) {
    System.out.println("3");
}
if (p2.equals(p3)) {
    System.out.println("4");
}
```



More text processing: Comparing strings

- There are more methods of a `String` object that can be used in ***<test>*** conditions.

Method	Description
`equals(str)`	whether this string contains exactly the same characters as the other string
`equalsIgnoreCase(str)`	whether this string contains the same characters as the other, ignoring upper- vs. lowercase differences
`startsWith(str)`	whether this string contains the other's characters at its start
`endsWith(str)`	whether this string contains the other's characters at its end

Comparing strings: Examples

- Hypothetical examples, assuming the existence of various `String` variables:

```
if (title.endsWith("M.D.")) {
    System.out.println("What's your number?");
}
```

```
if (fullName.startsWith("Marty")) {
    System.out.println("When's your 13th birthday?");
}
```

```
if (lastName.equalsIgnoreCase("lumBerg")) {
    System.out.println("I need your TPS reports!");
}
```

```
if (name.toLowerCase().indexOf("sr.") >= 0) {
    System.out.println("You must be old!");
}
```


`while` loops

Readings: 5.1

Definite loops

- **definite loop**: A loop that executes a known number of times.
  - The `for` loops we have seen so far are definite loops.
- We often use language like
  - "Repeat these statements *N* times."
  - "For each of these 10 things, …"
- 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**: A loop where it is not obvious in advance how many times it will execute.
- We often use language like
  - "Keep looping *as long as* or *while* this condition is still true."
  - "Don't stop repeating *until* the following happens."
- Examples:
  - Print random numbers until a prime number is printed.
  - Continue looping while the user has not typed "n" to quit.

`while` loop

- **`while` loop**: A control structure that repeatedly performs a test and executes a group of statements if the test evaluates to true.
- `while` loop, general syntax:

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

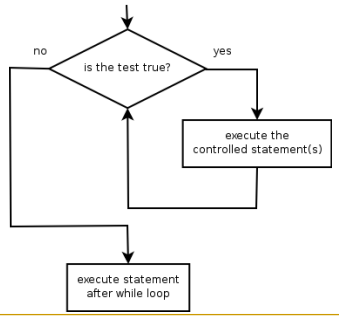
- Example:

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

Output:

```
1 2 4 8 16 32 64 128
```


`while` loop flow chart



Example

- Finds and prints 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);
```

Sample run:

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


`for` vs. `while`

- Any `for` loop of the following form:

```
for (<initialization>; <test>; <update>) {
    <statement(s)>;
}
```

is equivalent to a `while` loop of the following form:

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

`for` vs. `while`: Example

- What `while` loop is equivalent to the following `for` loop?

```
for (int i = 1; i <= 10; i++) {
    System.out.println(i + " squared = " + (i * i));
}
```

Solution:

```
int i = 1;
while (i <= 10) {
    System.out.println(i + " squared = " + (i * i));
    i++;
}
```


Exercise

- Write a program that will repeatedly prompt the user to type a number until the user types a non-negative number, then computes its square root.

Example log:

```
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: 11
The square root of 121 is 11.0
```


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 the `number` variable had to be declared outside the `while` loop in order to remain in scope.

Exercise: `digitSum`

- Write a method named `digitSum` that accepts an integer as a parameter and returns the sum of the digits of that number. You may assume that the number is non-negative.

Example:

`digitSum(29107)` returns 2+9+1+0+7 or 19

- Hint: Use the `%` operator to extract the last digit of a number. If we do this repeatedly, when should we stop?

Solution: `digitSum`

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


Sentinel loops

Readings: 5.1

Sentinel values

- **sentinel**: A special value that signals the end of the user's input.
- **sentinel loop**: A loop that repeats until a sentinel value is seen.
- Example: Write a program that repeatedly prompts the user for numbers to add until the user types 0, then outputs their sum. (In this case, 0 is our sentinel value.)

Sample run:

```
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 was 250
```


A 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 += number;
}

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

- Will this work? Why or why not?

Using a different sentinel value

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

Sample run:

```
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 was 250
```


Changing the sentinel value

- Just change the test 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 was " + sum);
```

- Now the solution produces the wrong output! Why?

```
The total was 249
```


The problem

- The current algorithm:

sum = 0.
while input is not the sentinel:
  prompt for input; read input.
  add input to the sum.

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

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

- What kind of problem is this?
  - This is a fencepost problem! We want to read *N* numbers (*N* is not known ahead of time), but only sum the first *N* - 1 of them.

Fencepost solution

- Here is a correct algorithm:

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

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

Sentinel solution

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

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


I hope you did not forget constants…

- An even better solution creates a constant for the sentinel. Why?

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

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

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


Indefinite loop variations

Readings: 5.4

Variant 1: `do/while`

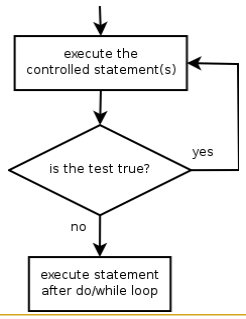
- **`do/while` loop**: A control structure that executes statements repeatedly while a condition is true, testing the condition at the end of each repetition.
- `do/while` loop, general syntax:

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

- Example:

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


`do/while` loop 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.

Variant 2: "Forever" loops

- Loops that go on… forever

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

- If it goes on forever, how do you stop?

`break`ing the cycle

- **`break` statement**: Immediately exits a loop (`for`, `while`, do/`while`).
- Example:

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

- Why is the `break` statement in an `if` statement?

Revisiting the sentinel problem

- Sentinel loop 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 += number;  // number != -1 here
}

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


Random numbers

Readings: 5.1

The `Random` class

- Objects of the `Random` class generate *pseudo-random* numbers.
  - Class `Random` is found in the `java.util` package.

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

- The methods of a `Random` object

Method name	Description
`nextInt()`	returns a random integer
`nextInt(`*max*`)`	returns a random integer in the range [0, *max*) in other words, from 0 to one less than *max*
`nextDouble()`	returns a random real number in the range [0.0, 1.0)

Generating random numbers

```
Random rand = new Random();
int randomNum = rand.nextInt(10);
// randomNum has a random value between 0 and 9
```

- What if we wanted a number from 1 to 10?

```
int randomNum = rand.nextInt(10) + 1;
```

- What if we wanted a number from *min* to *max* (i.e. an arbitrary range)?

```
int randomNum = rand.nextInt(<size of the range>) + <min>
```

where ***<size of the range>*** equals (***<max>*** `-` ***<min>*** `+ 1`)

Random questions

- Given the following declaration, how would you get:
  - A random number between 0 and 100 inclusive?
  - A random number between 1 and 100 inclusive?
  - A random number between 4 and 17 inclusive?

Random solutions

- Given the following declaration, how would you get:
  ```
  Random rand = new Random();
  ```
  - A random number between 0 and 100 inclusive?
    ```
    int random1 = rand.nextInt(101);
    ```
  - A random number between 1 and 100 inclusive?
    ```
    int random1 = rand.nextInt(100) + 1;
    ```
  - A random number between 4 and 17 inclusive?
    ```
    int random1 = rand.nextInt(14) + 4;
    ```


Exercise: Die-rolling

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

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


Solution: Die-rolling

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

public class Roll {
    public static void main(String[] args) {
        Random rand = new Random();

        int sum = 0;
        int tries = 0;
        while (sum != 7) {
            int roll1 = rand.nextInt(6) + 1;
            int roll2 = rand.nextInt(6) + 1;
            sum = roll1 + roll2;
            System.out.println("Roll: " + roll1 + " + " + roll2 + " = " + sum);
            tries++;
        }

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


Boolean logic

Readings: 5.2

True or false?

- **`boolean`**: A primitive type to represent logical values.
  - A `boolean` expression produces either `true` or `false`.
  - The ***<test>*** in `if/else` statements, `for` loops, and `while` loops are `boolean` expressions.
- Like any other type, you can create variables, parameters, and returns of type `boolean`.
- Examples:
  ```
  boolean minor = (age < 21);
  boolean iLoveCS = true;

  if (minor) {
      System.out.println("You can't purchase alcohol!");
  }
  ```


Logical operators

- Boolean expressions can use *logical operators*

Operator	Description	Example	Result
`&&`	and	`(9 != 6) && (2 < 3)`	`true`
`||`	or	`(2 == 3) || (-1 < 5)`	`true`
`!`	not	`!(7 > 0)`	`false`

Truth tables

- *Truth tables* of each operator used with `boolean` values p and q

p	q	p && q	p || q
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

p	!p
true	false
false	true

Boolean expressions

- What is the result of each of the following expressions?

```
int x = 42;
int y = 17;
int z = 25;
```

  - `y < x && y <= z`
  - `x % 2 == y % 2 || x % 2 == z % 2`
  - `x <= y + z && x >= y + z`
  - `!(x < y && x < z)`
  - `(x + y) % 2 == 0 || !((z - y) % 2 == 0)`

- Answers: `true`, `false`, `true`, `true`, `false`

Boolean methods

- There are methods that return `boolean` values.

Example:

```
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!");
}
```


Boolean methods

- Methods can return a `boolean` result.

```
public static boolean isLowerCaseLetter(char ch) {
    if ('a' <= ch && ch <= 'z') {
        return true;
    } else {
        return false;
    }
}
```

- Example usage:

```
String name = "e.e. cummings";
char firstLetter = name.charAt(0);
if (isLowerCaseLetter(firstLetter)) {
    System.out.println("You forgot to capitalize your name!");
}
```


Boolean "Zen"

- Methods that return a `boolean` result often have an `if/else` statement:

```
public static boolean isLowerCaseLetter(char ch) {
    if ('a' <= ch && ch <= 'z') {
        return true;
    } else {
        return false;
    }
}
```

- ... but the `if/else` is sometimes unnecessary.
  - The ***<test>*** is a `boolean` expression; its `true`/`false` value is exactly the value you want to return… so why not just return it directly!

```
public static boolean isLowerCaseLetter(char c) {
    return ('a' <= c && c <= 'z');
}
```


Exercises

- Write a method named `isVowel` that returns whether a particular character is a vowel (a, e, i, o, or u). Count only lowercase vowels.
  - `isVowel('q')` returns `false`
  - `isVowel('e')` returns `true`
- Write a method named `allDigitsOdd` that returns whether every digit of an integer is an odd number.
  - `allDigitsOdd(19351)` returns `true`
  - `allDigitsOdd(234)` returns `false`
- Write a method named `countVowels` that returns the number of lowercase vowels in a `String`.
  - `countVowels("Marty Stepp")` returns 2
  - `countVowels("e pluribus unum")` returns 6

Exercise

- Write a program that compares two words typed by the user to see whether they "*rhyme*" (end with the same last two letters) and/or *alliterate* (begin with the same letter)

Sample runs:

(run #1)
```
Type two words: car STAR
They rhyme!
```

(run #2)
```
Type two words: bare bear
They alliterate!
```

(run #3)
```
Type two words: sell shell
They alliterate!
They rhyme!
```


Solution

```
import java.util.*;

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

        System.out.print("Type two words: ");
        String word1 = console.next();
        String word2 = console.next();

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

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

    ...
```


Solution

```
    // Checks whether two words have the same last two letters,
    // ignoring case.  Assumes that the words have at least two
    // letters.
    public static boolean doesRhyme(String word1, String word2) {
        int len1 = word1.length();
        int len2 = word2.length();

        String last1 = word1.substring(len1 - 2, len1);
        String last2 = word2.substring(len2 - 2, len2);

        return last1.equalsIgnoreCase(last2);
    }

    // Checks whether two words start with the same letter,
    // ignoring case.
    public static boolean doesAlliterate(String word1, String word2) {
        // make both words lower-case, in case one word
        // starts with a capital letter and the other
        // one doesn't
        word1 = word1.toLowerCase();
        word2 = word2.toLowerCase();

        return (word1.charAt(0) == word2.charAt(0));
    }
}
```


Exercise

- Write a program that reads a number from the user and tells whether it is prime, and if not, gives the next prime after it.

Sample runs:

(run #1)
```
Type a number: 29
29 is prime
```

(run #2)
```
Type two numbers: 14
14 is not prime; the next prime after 14 is 17
```

- As part of your solution, you should write the following methods:
  - `isPrime`: Returns `true` if the parameter passed is a prime number.
  - `nextPrime`: Returns the next prime number whose value is greater than or equal to the parameter passed.

Solution

```
import java.util.*;

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

        System.out.print("Type a number: ");
        int num = console.nextInt();
        printPrimeStatus(num);
    }

    // prints that primality status of a number;  if
    // it's not prime, it computes the next one and prints
    // that too
    public static void printPrimeStatus(int num) {
        if (isPrime(num)) {
            System.out.println(num + " is prime");
        } else {
            System.out.println(num + " is not prime; the next prime after " + num +
                               " is " + nextPrime(num));
        }
    }

    ...
```


Solution

```
    // returns the next prime that is greater than
    // or equal to NUM
    public static int nextPrime(int num) {
        while (!isPrime(num)) {
            num++;
        }

        return num;
    }

    // returns true if the number given is a prime number
    public static boolean isPrime(int num) {
        if (num <= 1) {
            return false;
        }

        for (int i = 2; i < num; i++) {
            if (num % i == 0) {
                return false;
            }
        }

        return true;
    }
}
```

Exercise

- Modify your program from the previous slide so that it reads two numbers and tells whether each number is prime, and if not, gives the next prime after it; also tell whether they are *relatively prime* (i.e., have no common factors).

  Sample runs:

```
(run #1)
Type two numbers: 9 16
9 is not prime; the next prime after 9 is 11
16 is not prime; the next prime after 16 is 17
9 and 16 are relatively prime

(run #2)
Type two numbers: 7 21
7 is prime
21 is not prime; the next prime after 21 is 23
7 and 21 are not relatively prime
```


Solution

```
import java.util.*;

// note the code re-use from Primes.java
public class RelativePrimes {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);

        System.out.print("Type two numbers: ");
        int num1 = console.nextInt();
        int num2 = console.nextInt();

        printPrimeStatus(num1);
        printPrimeStatus(num2);
        printRelativePrimeStatus(num1, num2);
    }

    // prints whether the two given numbers are relatively prime to each other
    public static void printRelativePrimeStatus(int num1, int num2) {
        System.out.print(num1 + " and " + num2 + " are ");
        if (!areRelativelyPrime(num1, num2)) {
            System.out.print("not ");
        }
        System.out.println("relatively prime");
    }
```


Solution

```
// returns true if the given numbers are relatively prime
// (i.e. have no common factors)
public static boolean areRelativelyPrime(int num1, int num2) {
    for (int i = 2; i <= num1; i++) {
        // do numbers have a common factor?
        if ((num1 % i == 0) && (num2 % i == 0)) {
            return false;
        }
    }

    return true;
}

// prints that primality status of a number;  if
// it's not prime, it computes the next one and prints
// that too
public static void printPrimeStatus(int num) {
    if (isPrime(num)) {
        System.out.println(num + " is prime");
    } else {
        System.out.println(num + " is not prime; the next prime after " + num +
                           " is " + nextPrime(num));
    }
}
```


Solution

```
    // returns the next prime that is greater than
    // or equal to NUM
    public static int nextPrime(int num) {
        while (!isPrime(num)) {
            num++;
        }

        return num;
    }

    // returns true if the number given is a prime number
    public static boolean isPrime(int num) {
        if (num <= 1) {
            return false;
        }

        for (int i = 2; i < num; i++) {
            if (num % i == 0) {
                return false;
            }
        }

        return true;
    }
}
```


Exercise: Multiplication tutor

- Write a multiplication tutor program. Example log of execution:

```
This program helps you practice multiplication
by asking you random multiplication questions
with numbers ranging from 1 to 20
and counting how many you solve correctly.

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 correct answer was 266
You solved 5 correctly.
```

- Use a class constant for the maximum value of 20.

Solution: Multiplication tutor

```
import java.util.*;

// Asks the user to do multiplication problems and scores them.
public class MultTutor {
    public static final int MAX = 20;

    public static void main(String[] args) {
        introduction();
        Scanner console = new Scanner(System.in);

        // loop until user gets one wrong
        int correct = 0;
        while (askQuestion(console)) {
            correct++;
        }

        System.out.println("You solved " + correct + " correctly.");
    }

    ...
```

Solution: Multiplication tutor

```
    public static void introduction() {
        System.out.println("This program helps you practice multiplication");
        System.out.println("by asking you random multiplication questions");
        System.out.println("with numbers ranging from 1 to " + MAX);
        System.out.println("and counting how many you solve correctly.");
        System.out.println();
    }

    public static boolean askQuestion(Scanner console) {
        // pick two random numbers between 1 and 20 inclusive
        Random rand = new Random();
        int num1 = rand.nextInt(MAX) + 1;
        int num2 = rand.nextInt(MAX) + 1;

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


Reasoning about assertions

Readings: 5.5

Assertions

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

Examples:

- Java was created in 1995. (true)
- 10 is greater than 20. (false)
- Humphrey Bogart said "Play it again, Sam" in Casablanca. (false)
- Marty is 12. (true)
- x divided by 2 equals 7. (*depends on the value of x*)

Reasoning about assertions

- Suppose you have the following

```
if (x > 3) {
    // Point A: do something
} else {
    // Point B: do something else
}
```

- What do you know at the two different points?
  - Is *x > 3*? Always? Sometimes? Never?

Reasoning about assertions

```
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)
```


Debugging with reasoning: What's wrong?

```
import java.util.*;

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

        String password = "password";

        System.out.print("Enter password: ");
        String input = console.next();
        while (input != password) {
            System.out.println("Wrong password!");
            System.out.print("Enter password: ");
            input = console.next();
        }
    }
}
```

Strings are objects—should not use !=

```
import java.util.*;

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

        String password = "password";

        System.out.print("Enter password: ");
        String input = console.next();
        while (!input.equals(password)) {
            System.out.println("Wrong password!");
            System.out.print("Enter password: ");
            input = console.next();
        }
    }
}
```


Assertions: Example 1

```
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;
}
```

	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

Assertions: Example 2

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

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

        // Point C
        z++;

        // Point D
    }

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

	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

Assertions: Example 3

```
// pre : y >= 0,   post: 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;
            y /= 2;

            // Point D
        } else {
            // Point E
            prod *= x;
            y--;
            // Point F
        }
        // Point G
    }
    // Point H
    return prod;
}
```

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