What we will cover:
Naming Conventions
char and String Data Types, Unicode, ASCII code
Type Conversion
Mathematical operations, Math Class

Class Exercise 7 – Convert Kilometers per hour to Miles per hour

Naming Conventions
Use descriptive, explanatory names for identifiers in your program. Make sure that they are descriptive, but concise; this is a hard balance to achieve and takes practice and experience. Names are case sensitive and recall that there are different conventions for classes, variables, methods, and constants.

Methods and variables should start with a capital letter and use mixed-case capitalization if made up of multiple words. Ex: int reallyLongIdentifierIsTooLong; or public void methodNameIsWayTooLong(...). Following our descriptive but concise rule, better examples might be int maximumValue or public double computeArea(...).

Capitalize the first letter of class names. public class SimpleJava {

Constants should have all letters capitalized and use underscores if multiple words: PI and MIN_VALUE.

Brace/indent style fights falls into two major camps: end-of-line and next-line brace style. This refers to where the opening brace ( {  ) is when starting a code block. Java API source code (along with the book) is written in end-of-line style which claims to promote space savings and reduce confusion about blocks. An example:

    public class AdditionSimple {
        public static void main(String[] args) {
            ...
        }
    }

Next-line style claims to promote readability and is the standard for Visual Studio languages, including C#. An example:

    public class AdditionSimple
    {
        public static void main(String[] args)
        {
            ...
        }
    }

This was much more of a debate while developing on terminals that could only display 24 lines at a time. It really does not matter which style you choose to use, just be consistent and do not mix brace styles.

char and String Data Types
The character data type can represent a single character (letters, digits, special characters). Character literals must be enclosed in single quotation marks. For example

    char letter = 'a';
    char digit = '5';
    char special = '!';

In addition to printable characters, escape sequences can be used to represent special characters such as tabs and carriage returns. An escape sequence is a backslash (\) followed by a character.
<table>
<thead>
<tr>
<th>Description</th>
<th>Escape Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>backspace</td>
<td>\b</td>
</tr>
<tr>
<td>tab</td>
<td>\t</td>
</tr>
<tr>
<td>line feed</td>
<td>\n</td>
</tr>
<tr>
<td>backslash</td>
<td>\</td>
</tr>
<tr>
<td>single quote</td>
<td>\’</td>
</tr>
<tr>
<td>double quote</td>
<td>\&quot;</td>
</tr>
</tbody>
</table>

For instance, you could space a table using the tab character:
```
System.out.println("Full Name\t Student ID\t Address\t");
System.out.println("John Smith\t 123456789\t Anywhere, GA 55555");
```
prints out:
```
Full Name       Student ID     Address
John Smith      123456789      Anywhere, GA 55555
```

The `String` data type is a predefined class, not a primitive data type. Strings are groups (technically, arrays) of characters. String literals must be enclosed in double quotation marks. For example:
```
String sentence;
sentence = "Hello there.";
```

Individual characters can be extracted from strings using the `charAt` method of the `String` class. This variable will be set to the character ‘e’.
```
char characterE = sentence.charAt(1);  // Java arrays start at element 0
```

Strings can be combined using the concatenation (+) operator, the statement
```
String sentence = "Hi" + " there" + " Bob.";
```
creates the string “Hi there Bob.” Take note of the spaces inside the double quotes along with the words “there” and “Bob.”

As we noted in the GUI input examples, `Strings` can be converted into integers and doubles using methods of the `Integer` and `Double` wrapper classes that are in the `java.lang` package:
```
String string_var = JOptionPane.showInputDialog(null, "Enter a number");
int int_var = Integer.parseInt(string_var);
double double_var = Double.parseDouble(string_var);
```

There are similar conversions for `byte`, `short`, `long`, `float`, `char`, and `boolean` types. See the `java.lang.String API` (http://download-llnw.oracle.com/javase/6/docs/api/) for all functions; choose `java.lang` in the package pane on the top-left, then `String` in the classes pane on the bottom-left.

**Unicode and ASCII Code**

Characters are represented in memory by a binary code. The American Standard Code for Information Interchange (ASCII) uses 7-bits (8 bits for extended ASCII) to represent characters. Java uses Unicode, a 16-bit coding scheme, to represent characters. Unicode values are indicated in Java by \u followed by a 4 digit hexadecimal number specifying the character. Since ASCII is a 7-bit code, up to $2^7 = 128$ characters can be represented. Unicode can represent $2^{16} = 65536$ characters. The first 128 codes (\u0000 to \u007F) in Unicode correspond to the 128 ASCII codes.

Character values can be specified as literals, escape sequences, or in Unicode. For examples:
```
char alpha = '\u03AC';
char dollarSign = '\u0024';
// Hi there bob linefeed, bob in Greek letters
String phrase = "Hi there \u0392 \u03BF \u0392 \n";
```
**Type Conversion**

Numeric type conversion from smaller range data types to larger range data types is done implicitly (automatically) by Java; a small piece of data can always fit into a space designated for a large piece of data with no loss of precision. If one of the operands is a double then the other operand is converted to a double; otherwise, if one of the operands is a float then the other operand is converted to a float; otherwise, if one of the operands is a long then the other operand is converted to a long; otherwise both operands are converted to int.

For example:

```java
float resf = 0.0F;
double resd = 0.0D;
resf = 9/5;
resd = 9.0/5;
System.out.println("resf: " + resf + "resd: " + resd);
```

outputs:

resf: 1.0
resd: 1.8

Type casting (explicit casting) allows you to specify how the type conversion should be done. This is needed if you want to cast a data type to a smaller-ranged data type. To cast a value of one data type to another you give the new data type in parenthesis preceding the value to be cast:

```java
(int) num1;
float num2;
double num3 = 75.25D;
num2 = (float) num3; // represented in less precision
num1 = (int) num3;  // fractional part is truncated
System.out.println("num1: " + num1 + " num2: " + num2);
```

Prints:

num1: 75 num2: 75.25

Casting does not affect the variable being cast. In the above example, `num3` is not changed. Characters can be cast into numeric types. Their Unicode value is what is stored into the numeric type. If the Unicode value fits in the data type implicit casting may be used, otherwise explicit casting must be used. Integers between 0x0000 and 0xFFFF can be cast into characters implicitly. They are converted into the equivalent Unicode character. For floating point numbers, the value is first cast into an `int` and then into a `char`.

**Java Math Class**

Most commonly used math functions are predefined in the Math class in java.lang. Some of the Math class data and methods are:

<table>
<thead>
<tr>
<th>Constants</th>
<th>Usage</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>pi</td>
<td>Math.PI</td>
<td>area = radius * radius * Math.PI</td>
</tr>
<tr>
<td>e</td>
<td>Math.E</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Methods</th>
<th>Usage</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute value</td>
<td>Math.abs()</td>
<td>res = Math.abs(-5);</td>
</tr>
<tr>
<td>Cosine</td>
<td>Math.cos()</td>
<td>res = Math.cos(0.0);</td>
</tr>
<tr>
<td>Sine</td>
<td>Math.sin()</td>
<td>res = Math.sin(Math.PI);</td>
</tr>
<tr>
<td>Minimum</td>
<td>Math.min()</td>
<td>res = Math.min(2.1, 5.7);</td>
</tr>
<tr>
<td>Maximum</td>
<td>Math.max()</td>
<td>res = Math.max(2.1, 5.7);</td>
</tr>
<tr>
<td>Power</td>
<td>Math.pow()</td>
<td>res = Math.pow(5.0,2.5);</td>
</tr>
<tr>
<td>Rounding</td>
<td>Math.round();</td>
<td>res = Math.round(5.75);</td>
</tr>
<tr>
<td>Square Root</td>
<td>Math.sqrt()</td>
<td>res = Math.sqrt(9.0);</td>
</tr>
<tr>
<td>Exponential</td>
<td>Math.exp()</td>
<td></td>
</tr>
</tbody>
</table>
Many of the Math Class methods are overloaded to work on `int`, `long`, `float`, and `double`. The sine and cosine methods expect the argument in radians. See the `java.lang.Math` API (http://download-llnw.oracle.com/javase/6/docs/api/) for all functions; choose `java.lang` in the package pane on the top-left, then `Math` in the classes pane on the bottom-left.

**Shortcut Operators**

Java has several operators that can be combined with assignment operations. The arithmetic operator is placed between the object on the left hand side of the assignment operator and the assignment operator to combine the arithmetic operation and the assignment operation. The operator will be applied to the current value of the object on the left hand side of the `=` operator and the expression on the right hand side of the `=` operator and the result is placed in the object on the left hand side of the `=` operator.

Java has shortcut operations for addition and assignment (`+=`), subtraction and assignment (`-=`), multiplication and assignment (`*=`), division and assignment (`/=`), and modulus and assignment (`%=`). The general format for shortcut operations is:

```java
<object> operator= <expression>;
```

For example:

```java
sum += number; // equivalent to sum = sum + number;
```

Java also has shortcut operators for incrementing (`++`) and decrementing (`--`). Increment and decrement are equivalent to adding or subtracting one from the current value. The increment and decrement operators can be placed before (pre-increment/decrement) or after (post-increment/decrement) the object. When used as just a statement, there is no difference between pre- and post-incrementing/decrementing, however, when used in an expression there is a difference in how the items are evaluated.

For example:

```java
sum++; // equivalent to sum = sum + 1;
++sum;
```

However, the two statements:

```java
sum = num++;  // not equivalent to sum = sum + 1;
sum = ++num;
```

are not equivalent.

The first is executed as:

```java
sum = num;
num = num + 1;
```

and the second as:

```java
num = num + 1;
sum = num;
```

When post-increment/decrementing, the increment or decrement is done after (post) the value is used in the expression. The current value is used, and then the increment/decrement is executed. When pre-increment/decrementing, the increment/decrement is executed first, then used in the expression.

In the previous example, if `num` was originally 3, the result of `sum = num++` would give `sum = 3`, and the result of `sum = ++num;` would give `sum = 4`;

**Class Exercise 8 – Distance between two points**