Conditionals, Boolean Logic and Flow Control

Flow Control with Decision Tree

Java supports controlling the flow of a program with a standard **if - else if - else** block. Be sure to use **{ }** carefully. The rules are:

1. **If the code block is a single statement, no curly braces are needed**
2. **If the code block contains multiple statements then curly braces are needed**

Use the **if – else if** tree if you have a set conditions that are mutually exclusive (only one match out of a collection of matches)

```
if(condition1){
    // code block will execute if condition is true
    // multiple statements must be contained in { }

} //end if
else if(condition2){
    // code block will execute if condition1 is false
    // and condition2 is true

} // end else if
else if(condition3){
    //code block will execute if conditions 1 and 2 are false
    // and condition3 is true

} // end else if
else{
    // code block will execute if conditions 1, 2 and 3 are false

} // end else
```

Java supports nesting of if statements. Pay close attention to the curly braces as they can become difficult to track if you do not implement proper indentation and line breaks.

Remember: Java does not force indentation upon you like Python. You need to develop the good habits of proper indentation, line breaking and white space. **Write readable code!!**

```
if(condition1){
    // code block will execute if condition is true
    // multiple statements must be contained in { }
    if(condition1_1){
        // code block will execute if condition1 is true
        // and condition1_1 is true
        if(condition1_1_1){
            // code block will execute if condition1 is true
            // and condition1_1 is true
            // and condition1_1_1 is true
        }   // end second level nesting
    }   // end condition 1_1 if
    else if(condition1_2){
        // code block will execute if condition1 is true
        // and condition1_1 is false
        // and condition 1_2 is true
    }   // end condition1_2 if
    else{
        // code block will execute if condition1 is true
        // and condition1_1 is false
        // and condition1_2 is false
    }   // end nested else

} //end condition1 if
```

Equality and Relational Operators

The equality and relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand. The majority of these operators will probably look familiar to you as well. Keep in mind that you must use "==", not "=", when testing if two primitive values are equal.

==	**equal to**
!=	**not equal to**
>	**greater than**
>=	**greater than or equal to**
<	**less than**
<=	**less than or equal to**

The following program ***ComparisonDemo.java*** illustrates these operators in conjunction with if statements. The program also includes a decision to use command line arguments if they exist. There is also a try/catch pattern to prevent the program from crashing if the command line arguments are not cast-able to ints. The following screen shot focuses on the decision structure. The full program follows.

```
    // are they equal?
    if(val1 == val2){
        System.out.println(val1 + " == " + val2);
        System.out.println("Only use == on primitive values");
        System.out.println("Use { } if code block contains multiple lines.");
        System.out.println("Otherwise the { } are optional");
    }
    else if(val1 != val2)
        // no { } required here. Only a single line of code in the block
        System.out.println( val1 + " does not equal " + val2);
    else if(val1 > val2)
        System.out.println(val1 + " is larger than " + val2);
    else if(val1 < val2)
        System.out.println(val1 + " is smaller than " + val2);
    else if(val1 <= val2)
        System.out.println(val1 + " may be smaller then or equal to " + val2);
}
```

Full program listing. This example illustrates the processing of command line arguments as integers. Remember, they enter the program as Strings.

```java
import java.util.InputMismatchException;

public class ComparisonDemo {

    Run | Debug
    public static void main(String[] args){
        int val1 = 1;
        int val2 = 2;

        // do we have command line args? If so use those
        if(args.length == 2){
            // args are Strings and must be converted. The data may not be proper integers, so use try/catch
            try{
                // use the static method parseInt from the Integer class to convert
                val1 = Integer.parseInt(args[0]);
                val2 = Integer.parseInt(args[1]);
            }catch(InputMismatchException ime){
                System.out.println("There was a problem with the command line args. Using literals instead");
            }
        }

        // are they equal?
        if(val1 == val2){
            System.out.println(val1 + " == " + val2);
            System.out.println("Only use == on primitive values");
            System.out.println("Use { } if code block contains multiple lines.");
            System.out.println("Otherwise the { } are optional");
        }
        else if(val1 != val2)
            // no { } required here. Only a single line of code in the block
            System.out.println( val1 + " does not equal " + val2);
        else if(val1 > val2)
            System.out.println(val1 + " is larger than " + val2);
        else if(val1 < val2)
            System.out.println(val1 + " is smaller than " + val2);
        else if(val1 <= val2)
            System.out.println(val1 + " may be smaller then or equal to " + val2);
    }
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ javac ComparisonDemo.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java ComparisonDemo 3 10
3 does not equal 10
```

This example uses the default values instead of command line arguments

```java
import java.util.InputMismatchException;

public class ComparisonDemo {

    Run | Debug
    public static void main(String[] args){
        int val1 = 1;
        int val2 = 2;

        // do we have command line args? If so use those
        if(args.length == 2){
            // args are Strings and must be converted. The data may not be proper integers, so use try/catch
            try{
                // use the static method parseInt from the Integer class to convert
                val1 = Integer.parseInt(args[0]);
                val2 = Integer.parseInt(args[1]);
            }catch(InputMismatchException ime){
                System.out.println("There was a problem with the command line args. Using literals instead");
            }
        }

        // are they equal?
        if(val1 == val2){
            System.out.println(val1 + " == " + val2);
            System.out.println("Only use == on primitive values");
            System.out.println("Use { } if code block contains multiple lines.");
            System.out.println("Otherwise the { } are optional");
        }
        else if(val1 != val2)
            // no { } required here. Only a single line of code in the block
            System.out.println( val1 + " does not equal " + val2);
        else if(val1 > val2)
            System.out.println(val1 + " is larger than " + val2);
        else if(val1 < val2)
            System.out.println(val1 + " is smaller than " + val2);
        else if(val1 <= val2)
            System.out.println(val1 + " may be smaller than or equal to " + val2);
    }
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ javac ComparisonDemo.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java ComparisonDemo
1 does not equal 2
```

What if we wanted to further classify these numbers in more detail than just “not being equal”. The relationship of 3 and 10 could be classified in the following way

3 does not equal 10
3 is smaller than 10
3 is smaller than or equal to 10

Using the exclusive decision tree of **if – else if – else if** would not support this approach. We would need to implement a non-exclusive **if – if – if** structure. This example ***ComparisonDemo2.java*** follows. Pay close attention to lines 22 – 36 in ***ComparisonDemo*** and ***ComparisonDemo2*** as well as the output generated by both examples. Notice how the non-exclusive logic in ***ComparisonDemo2*** allows multiple if blocks to execute, while the exclusive logic in ComparisonDemo1 only allows a single if block to execute.

```java
import java.util.InputMismatchException;

public class ComparisonDemo2 {

    Run | Debug
    public static void main(String[] args){
        int val1 = 1;
        int val2 = 2;

        // do we have command line args? If so use those
        if(args.length == 2){
            // args are Strings and must be converted. The data may not be proper integers, so use try/catch
            try{
                // use the static method parseInt from the Integer class to convert
                val1 = Integer.parseInt(args[0]);
                val2 = Integer.parseInt(args[1]);
            }catch(InputMismatchException ime){
                System.out.println("There was a problem with the command line args. Using literals instead");
            }
        }

        // are they equal?
        if(val1 == val2){
            System.out.println(val1 + " == " + val2);
            System.out.println("Only use == on primitive values");
            System.out.println("Use { } if code block contains multiple lines.");
            System.out.println("Otherwise the { } are optional");
        }
        if(val1 != val2)
            // no { } required here. Only a single line of code in the block
            System.out.println( val1 + " does not equal " + val2);
        if(val1 > val2)
            System.out.println(val1 + " is larger than " + val2);
        if(val1 < val2)
            System.out.println(val1 + " is smaller than " + val2);
        if(val1 <= val2)
            System.out.println(val1 + " is smaller than or equal to " + val2);
    }
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ javac ComparisonDemo2.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java ComparisonDemo2 3 10
3 does not equal 10
3 is smaller than 10
3 is smaller than or equal to 10
```

Strings in More Detail

Java does not support the easy string indexing and slicing that exists in Python. Strings in Java do provide myriad methods for accomplishing the same tasks though. For example, the String class provides a method named **charAt**, which extracts a character. It returns a **char**, a primitive type that stores an individual character (as opposed to strings of them).

```
jshell> String fruit = "banana";
   ...> char letter = fruit.charAt(0);
fruit ==> "banana"
letter ==> 'b'
```

The argument 0 means that we want the letter at position 0. Like array indexes, string indexes start at 0, so the character assigned to letter is '**b**'. Char variables in Java are primitives and can be tested with == like other primitive values.

```
jshell> if(fruit.charAt(0) == 'b')
   ...> System.out.println(fruit + " begins with a b");
banana begins with a b
```

The String API spec is expansive. Familiarize your self with it.

https://docs.oracle.com/javase/9/docs/api/java/lang/String.html

String Equality

Something strange happens when we try to test for String equality though.

```
1   public class StringEquality{
2
        Run | Debug
3       public static void main(String[] args){
4
5           // create two identical String objects using "new"
6           String string1 = new String("Hello");
7           String string2 = new String("Hello");
8
9           // test String equality using ==
10          System.out.print(string1 + " equals " + string2 + ": ");
11          System.out.println(string1 == string2);
12
13          // use shortcut String creation
14          String string3 = "Hello";
15          String string4 = "Hello";
16
17          // test String equality using ==
18          System.out.print(string3 + " equals " + string4 + ": ");
19          System.out.println(string3 == string4);
20      }
21  }
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ javac StringEquality.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java StringEquality
Hello equals Hello: false
Hello equals Hello: true
```

Why is Java returning **false** when comparing 2 String objects that are identical? And then turning around and returning **true** when comparing 2 identical String objects? This behavior is based on the way the JVM handles the creation of String objects and requires some explanation.

The JVM, immutable Strings, String interning and the String Pool

The String object is the most used class in the Java language. Due this abundance of usage, the Java developers made some interesting decisions on how to handle the creation and storage of String objects within the JVM. The following discussion will focus on the special memory area within the JVM called the **string pool** .

String Interning

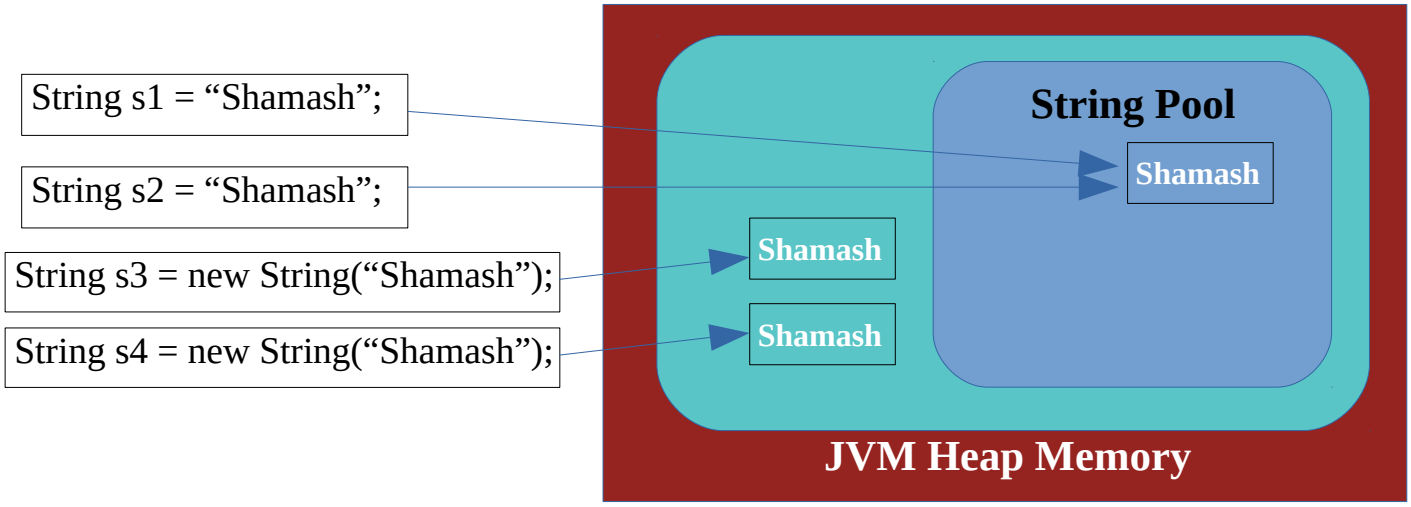
Thanks to the immutability of Strings in Java, the JVM can optimize the amount of memory allocated for them by storing only one copy of each String literal in the pool. This process is called **interning**. When we create a String variable and assign a value to it, the JVM searches the pool for a String of equal value. If found, the Java compiler will simply return a reference to its memory address, without allocating additional memory. If not found, it'll be added to the pool (interned) and its reference will be returned. Let's write a small test to verify this:

```
jshell> String s1 = "Shamash";
   ...> String s2 = "Shamash";
s1 ==> "Shamash"
s2 ==> "Shamash"

jshell> s1 == s2;
$78 ==> true
```

Strings allocated using the constructor: When creating a String via the new operator, the compiler will ***always create a brand spanking new object***. This object will be stored in the heap space reserved for the JVM. Every String variable created in this way will ***point to a unique address***. Remember that the equality operator == operates on primitive values. When you use this operator to test two **objects** the value that is being tested if the **reference** being held by the variables. It does not test the actual value of the String characters. This explains why the example above works. When using the shortcut String creation syntax a new String **will not be created if it has already been created.** Therefore the JVM **returns the reference** to the duplicate String object.

The following code snippet proves this.

```
jshell> String s1 = new String("Hello");
   ...> String s2 = new String("Hello");
s1 ==> "Hello"
s2 ==> "Hello"

jshell> s1 == s2;
$75 ==> false
```



JVM Memory

The String Pool

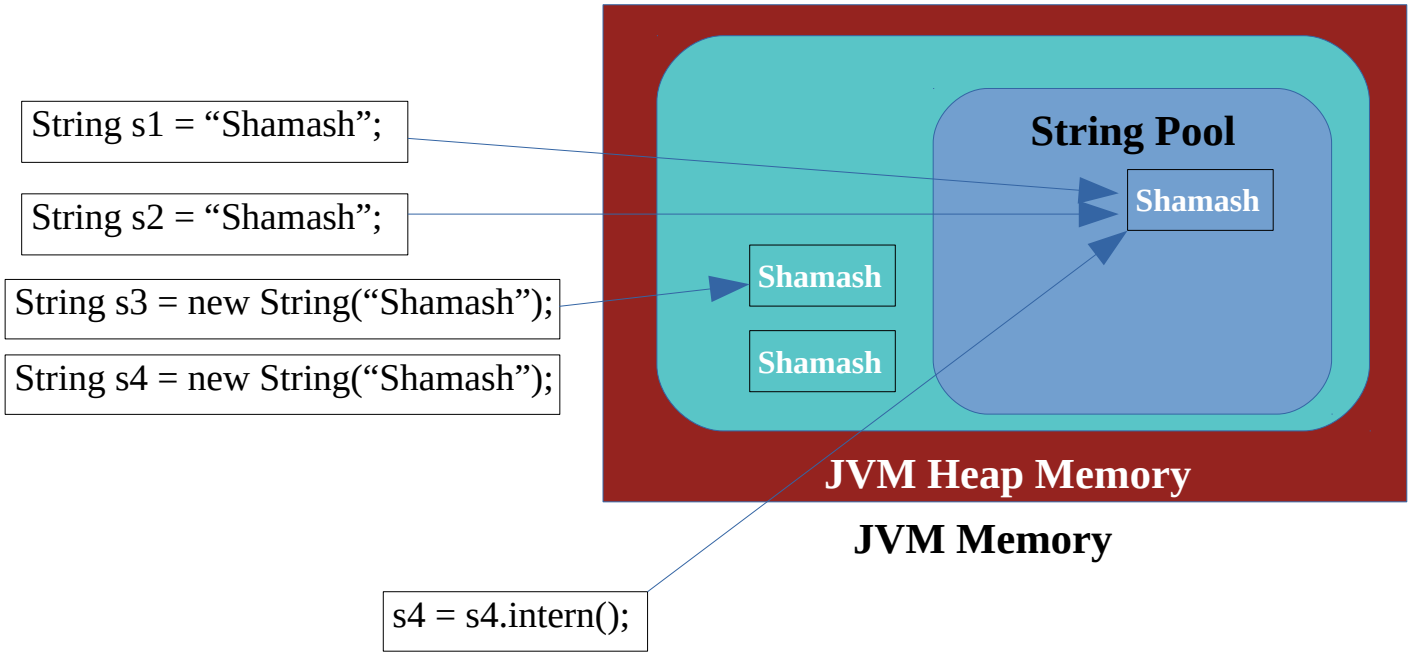
1. When we create the first string s1, there is no string with the value “Shamash” in the string pool. So the string “Shamash” is created in the pool and its reference is assigned to s1.
2. When we create the second string s2, there is already a string with the value “Shamash” in the string pool. The existing string reference is assigned to s2. References s1 and s2 both point to the same object
3. When we create third string s3, it’s created in the heap area because we are using the **new** operator.
4. When we create fourth string s4, it’s created in the heap area because we are using the **new** operator. We have created 4 string references but only 3 string objects.
5. When we compare **s1 == s2, it returns true** because both the variables are referring to the same string object.
6. When we compare **s3 == s4, it returns false** because they point to the different string objects. If we compared s3 or s4 to s1 or s2 this comparison would also return false.

The intern() Method

The String method **intern** gives programmers the ability to explicitly add String objects to the String pool. Closely analyze the following snippet

```
jshell> String s1 = new String("Shamash");
   ...> String s2 = "Shamash";
s1 ==> "Shamash"
s2 ==> "Shamash"

jshell> s1 == s2;
$92 ==> false

jshell> s1 = s1.intern();
s1 ==> "Shamash"

jshell> s1 == s2;
$94 ==> true
```

The line **s1 = s1.intern()** interns the String object pointed at by reference **s1,** adding it to the String pool. Because there was already a String object matching this pattern in the string pool, the JVM returns the reference to this object, saving us space. The references **s1 and s2** now point to the same object. Because String objects are immutable there is no chance that the String can become corrupted.



After calling **intern** on S4 the JVM sees that there is already a string object with the value “Shamash” in the String pool, so the reference to that object is returned and assigned to S4. So what happens to the String object that was referenced by s4? It gets **garbage collected** by the JVM. Java garbage collection is the process by which the JVM performs automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine. When Java programs run on the JVM, objects are

created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory. Java garbage collection is an **automatic process**. The programmer does not need to explicitly mark objects to be deleted. Objects are marked for garbage collection whenever there ceases to be a valid reference to the object. When the String object referenced by s4 prior to interning gets redirected to the String Pool object of the same value, the original "Shamash" object is marked for deletion. **This is not an instantaneous process!** The JVM will get to it as it works its way through the garbage queue.

Benefits of the String Pool

This certainly is a complex situation, so what are the benefits.

1. Java String Pool allows **caching** of string objects. This saves a lot of memory for JVM that can be used by other objects. As mentioned before, string objects are statistically the most numerous.
2. Java String Pool helps increase performance of the application because of reusability. It saves time creating a new string if there is already a string present in the pool with the same value. So there is an advantage in both **time and space** when pooling is used and because String objects are immutable, there is no worry about corruption.
3. **Best Practices:** Always create String objects in the String Pool

The String equals() method

So what is the most reliable way to determine the equality of String objects . . . ***or any object for that matter***? And to be clear, this concept has further repercussions than just String objects.

Shallow Equals

As we have seen, the equality operator == only works on primitive values, so when used with two object references the data that is being tested is the **address of the object.** A Java reference is a 32 bit integer which resolves to a memory address, so when we attempt a comparison like the following . . .

```
jshell> String s1 = new String("Shamash");
   ...> String s2 = new String("Shamash");
s1 ==> "Shamash"
s2 ==> "Shamash"

jshell> s1 == s2;
$3 ==> false
```

. . . it is the **32 bit integer reference** that is being tested. When used with objects, this type of comparison is called a **shallow equals** because the comparison is not looking at the internal composition of the objects, it is just looking at the reference values. Java does provide a method for us to perform a **deep equals** comparison on String objects . . . the **equals()** method. Here is the String API spec.

equals

`public boolean equals(Object anObject)`

Compares this string to the specified object. The result is `true` if and only if the argument is not `null` and is a `String` object that represents the same sequence of characters as this object.

For finer-grained String comparison, refer to `Collator`.

Overrides:
`equals` in class `Object`

Parameters:
`anObject` - The object to compare this `String` against

Returns:
`true` if the given object represents a `String` equivalent to this string, `false` otherwise

See Also:
`compareTo(String)`, `equalsIgnoreCase(String)`

Let's see this in practice. Pay close attention to the syntax of this method call. It requires two objects

1. The object through which the method is being invoked: **s1**
2. The object that is being passed as an argument to the method call: **s2**

```
jshell> String s1 = new String("Shamash");
   ...> String s2 = new String("Shamash");
s1 ==> "Shamash"
s2 ==> "Shamash"

jshell> s1 == s2;
$3 ==> false

jshell> s1.equals(s2);
$4 ==> true
```

This **equals()** method will become something that we will define for the objects that we define ourselves. Make sure you fully understand the difference between a shallow comparison and a deep comparison.

The Conditional Operators

The **&&** and || operators perform Conditional-AND and Conditional-OR operations on two boolean expressions. These operators exhibit "short-circuiting" behavior, which means that the second operand is evaluated only if needed.

&&	Conditional AND
||	Conditional OR
!	Logical Negation

The following program ***ConditionalDemo.java*** demonstrates these operators

```java
public class ConditionalDemo {

    Run | Debug
    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if((value1 == 1) && (value2 == 2))
            System.out.println("value1 is 1 AND value2 is 2");
        if((value1 == 1) || (value2 == 1))
            System.out.println("value1 is 1 OR value2 is 1");
    }
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ javac ConditionalDemo.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java ConditionalDemo
value1 is 1 AND value2 is 2
value1 is 1 OR value2 is 1
```

Another conditional operator is **`?:`**, which can be thought of as shorthand for an `if-then-else` statement. This operator is also known as the **ternary operator** because it uses three operands. In the following example, this operator should be read as:

"If someCondition is true, assign the value of value1 to result. Otherwise, assign the value of value2 to result."

```java
public class TernaryOperatorDemo{

    Run | Debug
    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        int result;
        boolean someCondition = value1 < value2;
        result = someCondition ? value1 : value2;

        System.out.println(result + " is the smallest value");
    }
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ javac TernaryOperatorDemo.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java TernaryOperatorDemo
1 is the smallest value
```

The Switch Statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case. The basic structure of a switch statement follows.

```
switch(expression) {

    case value :
          // Statements
          break; // optional

       case value :
          // Statements
          break; // optional

       // You can have any number of case statements.
       default : // Optional
          // Statements
}
```

The following rules apply to a switch statement −

The variable used in a switch statement can only be integers, convertible integers (byte, short, char), strings and enums (we will be covering enums).

You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.

When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.

When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.

A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Here is an example that resolves the month number to the month string. Notice that the flow of this program is analogous to

if month is 1 then the string is “January”

else if the month is 2 then the string is “February”

else if the month is 3 then the month is “March”

etc . . .

```java
public class SwitchDemo {
    Run | Debug
    public static void main(String[] args) {

        int month = 8;
        String monthString;
        switch (month) {
            case 1:  monthString = "January";
                     break;
            case 2:  monthString = "February";
                     break;
            case 3:  monthString = "March";
                     break;
            case 4:  monthString = "April";
                     break;
            case 5:  monthString = "May";
                     break;
            case 6:  monthString = "June";
                     break;
            case 7:  monthString = "July";
                     break;
            case 8:  monthString = "August";
                     break;
            case 9:  monthString = "September";
                     break;
            case 10: monthString = "October";
                     break;
            case 11: monthString = "November";
                     break;
            case 12: monthString = "December";
                     break;
            default: monthString = "Invalid month";
                     break;
        } // end switch
        System.out.println("Month number " + month + " is " + monthString);
    } // end main
} // end class
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ javac SwitchDemo.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java SwitchDemo
Month number 8 is August
```

Pay close attention to the **break** statements, as they are required to achieve the desired result. I will demonstrate what happens if you forget a break statement.
Note: The cases determine an entry point. Once the switch block is entered, the code will continue processing statements until

1. A break statement is encountered
2. The end of the switch statement is encountered.

Examine the same program with the break statements removed.

```
1   public class SwitchNoBreakDemo {
        Run | Debug
2       public static void main(String[] args) {
3
4           int month = 8;
5           String monthString;
6           switch (month) {
7               case 1:  monthString = "January";
8               case 2:  monthString = "February";
9               case 3:  monthString = "March";
10              case 4:  monthString = "April";
11              case 5:  monthString = "May";
12              case 6:  monthString = "June";
13              case 7:  monthString = "July";
14              case 8:  monthString = "August";
15              case 9:  monthString = "September";
16              case 10: monthString = "October";
17              case 11: monthString = "November";
18              case 12: monthString = "December";
19              default: monthString = "Invalid month";
20          } // end switch
21          System.out.println("Month number " + month + " is " + monthString);
22      } // end main
23  } // end class
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL                                  2:

kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ javac SwitchNoBreakDemo.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java SwitchNoBreakDemo
Month number 8 is Invalid month
```

Notice the output . . . **Month number 8 is Invalid month**. What happened here?

1. Case 8 was found to be the match. The match defines the entry point to the switch block.
2. Because there are no break statements, execution proceeds through case 9, 10, 11, 12 and default . . . overwriting the monthString variable with each month greater than 8.
3. The code completes when the send of the switch block is encountered on line 20 and monthString contains the last value assigned: **Invalid month**

This becomes more clear if we add print statements to each case.

```java
public class SwitchNoBreakDemo {
    Run | Debug
    public static void main(String[] args) {

        int month = 8;
        switch (month) {
            case 1:  System.out.println("January");
            case 2:  System.out.println("February");
            case 3:  System.out.println("March");
            case 4:  System.out.println("April");
            case 5:  System.out.println("May");
            case 6:  System.out.println("June");
            case 7:  System.out.println("July");
            case 8:  System.out.println("August");
            case 9:  System.out.println("September");
            case 10: System.out.println("October");
            case 11: System.out.println("November");
            case 12: System.out.println("December");
            default: System.out.println("Invalid month");
        } // end switch
    } // end main
} // end class
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ javac SwitchNoBreakDemo.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java SwitchNoBreakDemo
August
September
October
November
December
Invalid month
```

As I hope you may be thinking an array of Strings with the ***month_number -1*** as an index to the table would be a **much better** way of implementing this logic.

Repetition Statements

Java supports the following repetition statements

- while
- do . . . while
- for
- for each

We will cover the basic syntax of each along with some examples.

While Loop, Example 1: Blastoff . . . a while loop that counts backwards from 10 and then prints Blastoff!

```java
public class LoopingDemo{

    Run | Debug
    public static void main(String[] args){

        int n = 10;
        while(n > 0){
            System.out.println(n);
            n--;
        }
        System.out.println("Blastoff!");


    } // end main
} // end class
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ javac LoopingDemo.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java LoopingDemo
10
9
8
7
6
5
4
3
2
1
Blastoff!
```

While Loop, Example 2: The Hailstone Sequence: https://en.wikipedia.org/wiki/Collatz_conjecture.

```java
        int n = 30;
        System.out.print("Halistone sequence for " + n + ": ");
        while (n != 1) {
            System.out.print(n + " ");
            if (n % 2 == 0) n /= 2;
            else n = n * 3 + 1;
        }
        System.out.println(n+"\n");
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/flow_control$ javac LoopingDemo.java
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/flow_control$ java LoopingDemo
Halistone sequence for 30: 30 15 46 23 70 35 106 53 160 80 40 20 10 5 16 8 4 2 1
```

While Loop, Example 3: Generating a table of data

Loops are good for generating and displaying tabular data. Before computers were readily available, people had to calculate logarithms, sines and cosines, and other common mathematical functions by hand. To make that easier, there were books of tables where you could look up values of various functions. Creating these tables by hand was slow and boring, and the results were often full of errors. When computers appeared on the scene, one of the initial reactions was: “This is great! We can use a computer to generate the tables, so there will be no errors.”

That turned out to be true (mostly), but shortsighted. Not much later, computers were so pervasive that printed tables became obsolete. Even so, for some operations, computers use tables of values to get an approximate answer, and then perform computations to improve the approximation.

In some cases, there have been errors in the underlying tables, most famously in the table the original Intel Pentium used to perform floating-point division

https://en.wikipedia.org/wiki/Pentium_FDIV_bug

Although a “log table” is not as useful as it once was, it still makes a good example of iteration. The following loop displays a table with a sequence of values in the left column and their logarithms in the right column:

```
23          n = 1;
24          while (n < 10) {
25              System.out.printf("%d => %f \n", n, Math.log(n));
26              n++;
27          }
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ javac LoopingDemo.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java LoopingDemo
1 => 0.000000
2 => 0.693147
3 => 1.098612
4 => 1.386294
5 => 1.609438
6 => 1.791759
7 => 1.945910
8 => 2.079442
9 => 2.197225
```

While Loop, Example 4: Generate multiplication table using nested while loops

```
28          int n = 1;      // counter for outer loop
29          int i = 1;      // counter for inner loop
30          int limit = 6;  // compute up to here
31
32          // outer loop
33          while (i < limit){
34              // inner loop
35              while (n <= limit) {
36                  System.out.printf("%4d", n * i);
37                  n++;
38              } // end inner loop
39              n = 1;                  // reset n
40              System.out.println();   // print a line break
41              i++;                    // increment i
42          } // end outer loop
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ javac LoopingDemo.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java LoopingDemo
   1   2   3   4   5   6
   2   4   6   8  10  12
   3   6   9  12  15  18
   4   8  12  16  20  24
   5  10  15  20  25  30
```

Increase the limit to expand the table

```
28          int n = 1;      // counter for outer loop
29          int i = 1;      // counter for inner loop
30          int limit = 12; // compute up to here
31
32          // outer loop
33          while (i < limit){
34              // inner loop
35              while (n <= limit) {
36                  System.out.printf("%4d", n * i);
37                  n++;
38              } // end inner loop
39              n = 1;                  // reset n
40              System.out.println();   // print a line break
41              i++;                    // increment i
42          } // end outer loop
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ javac LoopingDemo.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java LoopingDemo
   1   2   3   4   5   6   7   8   9  10  11  12
   2   4   6   8  10  12  14  16  18  20  22  24
   3   6   9  12  15  18  21  24  27  30  33  36
   4   8  12  16  20  24  28  32  36  40  44  48
   5  10  15  20  25  30  35  40  45  50  55  60
   6  12  18  24  30  36  42  48  54  60  66  72
   7  14  21  28  35  42  49  56  63  70  77  84
   8  16  24  32  40  48  56  64  72  80  88  96
   9  18  27  36  45  54  63  72  81  90  99 108
  10  20  30  40  50  60  70  80  90 100 110 120
  11  22  33  44  55  66  77  88  99 110 121 132
```

If you are unsure what these loops are doing then I would recommend typing the code up and pasting it into the Java variant of **PythonTutor (**http://pythontutor.com/java.html#mode=edit**)** to step through line by line.

Here is an example: https://tinyurl.com/qo8z8j7

While Loop Example 5: Iterate through a file of IP logs to count the number of addresses from China. Print the country code and IP address also. Pass the file name as a command line argument. If no file name is passed, the program aborts.

There are two ways to read character based input from an input stream

- **Scanner:** The Scanner reads character based input streams and also parses and converts tokens using regular expression pattern matching. The Scanner uses a 1KB buffer. Scanner operations are not synchronized and thus not thread safe. There are many read methods

- **BufferedReader:** A BufferedReader is a simple class meant to efficiently read from the underling stream. Generally, each read request made of a Reader like a FileReader causes a corresponding read request to be made to underlying stream. Each invocation of read() or readLine() could cause bytes to be read from the file, converted into characters, and then returned, which can be inefficient. Efficiency is improved appreciably if a Reader is wrapped in a BufferedReader. A BufferedReader reads character based input from an input stream but performs no tokenized parsing (this is colloquially called **dumb reading**). You would then need to manually inspect the input. BufferedReader uses an 8KB buffer to cut down on disk operations. Operations on a BufferedReader are synchronized, meaning they are thread safe in a multi-threaded environment.

  There are two basic read methods:

  - read: Reads a single character
  - readLine: reads up until a line break

- Choose the approach that works best for your situation.

Important Note: Recent updates to Java have introduced a variety of streaming classes and methods and abstract many details of file reading. I will leave those for you to explore. I want to focus on mechanics here.

Code listings follow

Example of BufferedReader

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.FileReader;

public class BufferedReaderDemo{

    Run | Debug
    public static void main(String[] args){

        // are there command line args?
        if(args.length > 0) {
            String fileName     = "";
            String line         = "";
            int chinaCount      = 0;
            fileName            = args[0];

            try {
                FileReader fileReader = new FileReader(fileName);        //create a file reader object
                BufferedReader bufferedReader = new BufferedReader(fileReader);     // Buffer the file reader
                while (true) {  // BufferedReader has no boolean control methods. Must break internally
                    line = bufferedReader.readLine();                   // read a line
                    if (line == null) break;                            // if the line is null break out of loop
                    String country = line.substring(line.length() - 2); //extract the country code
                    if(country.equals("CN")){                           // China? then extract the address
                        chinaCount++;
                        // extract ip address
                        String[] tokens = line.split(",");              // split on commas
                        String ip = tokens[1];                          // ip address is 2nd element
                        System.out.println(country + " " + ip);
                    }
                }
                System.out.println("There were " + chinaCount + " ip addresses logged from China");
                bufferedReader.close();                                 // close the resource
            }catch(IOException ioe){
                System.out.println("There was a problem opening the file");
            }
        }else
            System.out.println("No file specified . . . aborting");
    } // end main
} // end class
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ javac BufferedReaderDemo.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java BufferedReaderDemo ip_logs.txt
CN 97.238.139.82
CN 75.15.98.143
CN 56.243.23.107
```

Syntactical things of note:

1. **Line 19:** BufferedReader does not have a boolean method to determine if there is more data to read. This means we have to test the input and if it is **null,** break out the loop (line 21). This must be done **before** parsing the data.

2. **Line 22:** Use the substring method and the length of the line to extract the Country code

3. **Line 26:** The String method split returns an array of String tokens. Here we are splitting on commas.

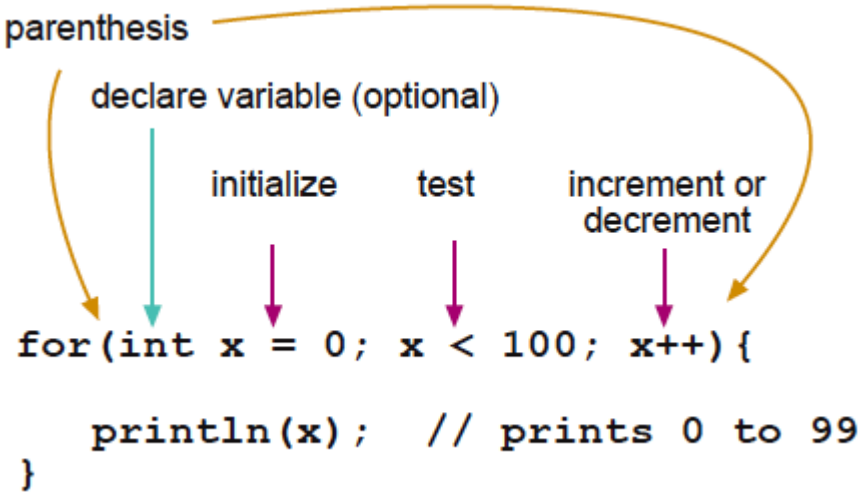
Example of Scanner

```java
import java.util.Scanner;
import java.io.IOException;
import java.io.FileReader;

public class ScannerFileDemo{

    Run | Debug
    public static void main(String[] args){

        // are there command line args?
        if(args.length > 0) {
            String fileName     = "";
            String line         = "";
            int chinaCount      = 0;
            fileName            = args[0];                          // get the file name from command line args

            try {
                FileReader fileReader       = new FileReader(fileName); // create a file reader object
                java.util.Scanner scanner   = new Scanner(fileReader);  // Scan the file reader
                scanner.useDelimiter(",");                              // have scanner use , as a delimiter

                while (scanner.hasNext()) {                             // use boolean method hasNext to control while loop
                    line = scanner.nextLine();                          // read a line
                    String country = line.substring(line.length() - 2); //extract the country code
                    if(country.equals("CN")){                           // China? then extract the address
                        chinaCount++;
                        // extract ip address
                        String[] tokens = line.split(",");              // split on commas
                        String ip = tokens[1];                          // ip address is 2nd element
                        System.out.println(country + " " + ip);
                    }
                }
                System.out.println("There were " + chinaCount + " ip addresses logged from China");
                scanner.close();                                        // close the resource
            }catch(IOException ioe){
                System.out.println("There was a problem opening the file");
            }
        }else System.out.println("No file specified . . . aborting");
    } // end main
} // end class
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ javac ScannerFileDemo.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java ScannerFileDemo ip_logs.txt
CN 97.238.139.82
CN 75.15.98.143
CN 56.243.23.107
```

For Loops:

The loops we have written so far have several elements in common. They start by initializing a variable, they have a condition that depends on that variable, and inside the loop they do something to update that variable. This type of loop is so common that there is another statement, the for loop, that expresses it more concisely. Here is the basic format of a Java for loop



for loops have three components in parentheses, separated by semicolons: the initializer, the condition, and the update.

1. The initializer runs once at the very beginning of the loop and establishes the initial condition of the counter.
2. The condition is checked each time through the loop. If it is false, the loop ends. Otherwise, the body of the loop is executed (again).
3. At the end of each iteration, the update runs, and we go back to step 2.

Let's re-write the multiplication table program using for loops. The code listing follows. Notice how compact the for loop syntax is compared to the while loop syntax. They both accomplish the same task with the same level of efficiency but the for loop is more readable.

```java
public class ForLoops{

    Run | Debug
    public static void main(String[] args){
        int limit = 5;

        if(args.length > 0){
            try{
                limit = Integer.parseInt(args[0]);
            }catch(NumberFormatException nfe){
                System.out.println("Invalid argument type. Using: " + limit);
            } // end catch
        } // end if

        System.out.println();

        for(int i = 1; i <= limit; ++i){
            for(int j = 1; j <= limit; ++j)
                System.out.printf("%4d", j * i);
            System.out.println();
        }// end outer for

        System.out.println();
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ javac ForLoops.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java ForLoops

   1   2   3   4   5
   2   4   6   8  10
   3   6   9  12  15
   4   8  12  16  20
   5  10  15  20  25

kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java ForLoops 10

   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
   6  12  18  24  30  36  42  48  54  60
   7  14  21  28  35  42  49  56  63  70
   8  16  24  32  40  48  56  64  72  80
   9  18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90 100
```

For loops and sequences

For loops are the perfect solution to iterating through a sequence such as an array or a String.

```
45          int count   = 0;
46          String text = "Shamash";
47
48          // use command line args if present
49          if(args.length > 0)
50              text = args[0];
51          // use the length of the String to control the loop
52          for(int index = 0; index < text.length(); ++index){
53              // String elements are primitive chars
54              char letter = text.charAt(index);
55              if(letter == 'a')
56                  count++;
57          }// end for
58
59          System.out.println("There are " + count + " 'a' letters in: " + text);
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                                2: Java

kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java ForLoops
There are 2 'a' letters in: Shamash
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java ForLoops prestidigitation
There are 1 'a' letters in: prestidigitation
```

Let's accomplish the same task but turn the String object into a char array using the String method **toCharArray**. This makes the element access more efficient, but does require the transformation of the String object. Notice the head scratching design to have the **String length be a method** while the **Array length is a variable.**

```
27          int count   = 0;
28          String text = "Shamash";
29
30          // use command line args if present
31          if(args.length > 0)
32              text = args[0];
33
34          // turn the String object into a primitive array of type char
35          char[] letters = text.toCharArray();
36
37          for(int index = 0; index < letters.length; ++index){
38              if(letters[index] == 'a')
39                  count++;
40          }
41          System.out.println("There are " + count + " 'a' letters in: " + text);
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                                2: Jav

kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ javac ForLoops.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java ForLoops
There are 2 'a' letters in: Shamash
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java ForLoops prestidigitation
There are 1 'a' letters in: prestidigitation
```

For Loop Example: You're working for the National Security Agency trying to decipher secret messages intercepted from foreign spies. We think that they are using letter pairs to hide data and we want to count the number of times paired letters occur in a source document of unknown size.

```
1   import java.io.FileReader;
2   import java.io.BufferedReader;
3   import java.io.IOException;
4
5   public class CountPairsInFile{
6
        Run | Debug
7       public static void main(String[] args){
8           String fileName = "";
9           String line     = "";
10          int count       = 0;
11
12          if(args.length > 0){
13              fileName = args[0];
14
15              try{
16                  FileReader fr = new FileReader(fileName);
17                  BufferedReader br = new BufferedReader(fr);
18                  // while loop will be terminated internally
19                  while(true){
20                      line = br.readLine();
21                      if(line == null) break;
22                      // loop through line counting pairs
23                      // more efficient than reading characters with the BufferedReader
24                      char[] chars = line.toCharArray();
25                      for(int index = 0; index < chars.length - 1; ++index){
26                          if(chars[index] == chars[index + 1]) count++;
27                      }
28                  }
29              }catch(IOException ioe){
30                  System.out.println("There was something wrong with the file");
31              }
32              System.out.println("There were " + count + " pairs in: " + fileName);
33              br.close();
34          } // end if
35          else System.out.println("No file name provided . . . aborting");
36      } // end main
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL                                  2: Java Proces

kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ javac CountPairsInFile.java
kenneth@dragonborn:~/Documents/Spring2020/165/examples/flow_control$ java CountPairsInFile intercepted.txt
There were 58 pairs in: intercepted.txt
```

Syntactical Notes

1. Read each line with a single BufferedReader readLine call. Once it's in RAM you can process the line with a for loop. This is more efficient than reading every character with a BufferedReader call as it cuts down on the disk activity.
2. Notice the for loop condition **(index < chars.length – 1).** Subtract one to cause the loop to go one less time due to the **+1** on line 26 . . . otherwise you will get a StringIndexOutOfBoundsException.

Black Bear Sightings: You are helping the forest service in Alaska to track the number of black bear sightings. You have a collection data from seven different regions. This data spans 12 months and you are given the totals for each month, from each region. You need to compute the following

1. Average black bear sightings per month
2. Average black bear sightings per region

The data has been exported to a comma separated file of the following structure. Each line in the file represents the data from a region. Each region has 12 data points, one for each month.

black_bear_sightings.txt

```
41,3,4,47,70,69,56,88,40,34,5,76
1,92,35,30,88,9,49,40,100,32,59,82
86,78,48,77,93,60,24,9,85,81,61,77
47,38,3,43,35,42,39,13,26,22,4,85
55,21,21,41,2,52,16,93,43,16,74,28
12,91,93,31,10,83,20,15,78,21,35,84
67,23,12,14,66,78,59,7,48,38,66,72
```

Design Choices

1. When the program launches read the data from the file into a 2 dimensional array and then close the file. All calculations will happen on the array.
2. Regions will be numbered as ***Region 1, Region 2, . . . Region N***
3. Months will be labeled by their name.

Operations

We now have a two dimensional table (matrix) and we need to process this table by column and by row. To get the average per month we need to sum each column. To get the average per region we need to sum each row.

Average for Region One is: 44.41

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
0	41	3	4	47	70	69	56	88	40	34	5	76
1	1	92	35	30	88	9	49	40	100	32	59	82
2	86	78	48	77	93	60	24	9	85	81	61	77
3	47	38	3	43	35	42	39	13	26	22	4	85
4	55	21	21	41	2	52	16	93	43	16	74	28
5	12	91	93	31	10	83	20	15	78	21	35	84
6	67	23	12	14	66	78	59	7	48	38	66	72

Average for January is: 44.14

Program Setup

- Lines 9 and 10 define constants used for dimensioning the array. This allows for easy change of table size in the future.
- Line 13 declares the 2D array using the constants
- Line 14 declares an array of Strings to hold the month names. This way we can easily extract the correct month with a loop index without any conditional logic.

```
5   public class BlackBears{
6
7       public static void main(String[] args){
8
9           final int NUM_REGIONS   = 7;
10          final int NUM_MONTHS    = 12;
11
12          String      fileName    = "";
13          int[][]     sightings   = new int[NUM_REGIONS][NUM_MONTHS];
14          String[]    months      = { "January", "February", "March", "April", "May",
15                                      "June", "July", "August", "September", "October",
16                                      "November", "December"};
17
```

- Line 18 makes sure the file name was provided when the program was launched
- Lines 22 and 23 create instances of the classes necessary to setup the Scanner and connect it to a file.
- The nested loops on lines 26 through 32 load the data from the file line by line. Each line is split into an array, which is then iterated through. The individual data items are placed into table cells using the **[row][column]** indexing idiom. Notice that the constants are used again to control the loop variables. Simply changing those constant values would allow the loops to expand or contract as needed.

```
18          if(args.length > 0){
19              fileName = args[0];
20
21              try{
22                  FileReader fr   = new FileReader(fileName);
23                  Scanner scanner = new Scanner(fr);
24
25                  // load the table with the sightings data
26                  for(int i = 0; i < NUM_REGIONS; i++){
27                      String line     = scanner.nextLine();
28                      String[] values = line.split(",");
29                      for(int j = 0; j < NUM_MONTHS; ++j){
30                          sightings[i][j] = Integer.parseInt(values[j]);
31                      } // end inner for
32                  } // end outer for
33
```

Average Sightings Per Month

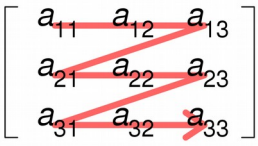
```
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/flow_control$ java BlackBears black_bear_sightings.txt
Average for January:     44.14
Average for February:    49.43
Average for March:       30.86
Average for April:       40.43
Average for May:         52.00
Average for June:        56.14
Average for July:        37.57
Average for August:      37.86
Average for September:   60.00
Average for October:     34.86
Average for November:    43.43
Average for December:    72.00
```

Months

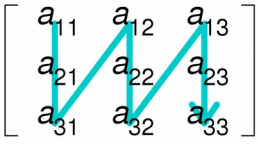
	0	1	2	3	4	5	6	7	8	9	10	11
0	41	3	4	47	70	69	56	88	40	34	5	76
1	1	92	35	30	88	9	49	40	100	32	59	82
2	86	78	48	77	93	60	24	9	85	81	61	77
3	47	38	3	43	35	42	39	13	26	22	4	85
4	55	21	21	41	2	52	16	93	43	16	74	28
5	12	91	93	31	10	83	20	15	78	21	35	84
6	67	23	12	14	66	78	59	7	48	38	66	72

January is month 0, February is month 1 etc. In order to accomplish this task we need to iterate through the array in **column major order;** processing the table one column at a time. This is different from how we loaded the table, which was in row **major order**. The difference between these two table iterations lies in the way we deal with our loop index variables.

Row-major order



Column-major order



Here is a review of the row major order code to fill the table. Notice that the outer loop is based on the number of regions. In the data set that is the row. One row for each region. Controlling the loop in this fashion means that we implement the outer loop with the major order dimension. The outer loop in this scenario will iterate 7 times. For each one of those steps, the inner loop will execute NUM_MONTHS times . . . or 12.

```
// load the table with the sightings data
for(int i = 0; i < NUM_REGIONS; i++){
    String line     = scanner.nextLine();
    String[] values = line.split(",");
    for(int j = 0; j < NUM_MONTHS; ++j){
        sightings[i][j] = Integer.parseInt(values[j]);
    } // end inner for
} // end outer for
```

To refactor this into column major order we simply reverse the structure of the for loops, placing the loop that goes NUM_MONTHS times with the loop that goes NUM_REGIONS times.

```
// compute average per month
for(int i = 0; i < NUM_MONTHS; ++i){
    double sum = 0.0;
    for(int j = 0; j < NUM_REGIONS; ++j)
        sum += sightings[j][i];
    System.out.printf("Average for %s:\t %.2f \n", months[i], sum / NUM_REGIONS);
}
```

Look carefully at this code because you also have to exchange the locations of loop control variables **i and j.**

- **Row Major Order:** [i][j]
- **Column Major Order:** [j][i]

Average Sightings Per Region

This is handled in simple row major order like the loading of the table. The outer loop control for the rows (regions) and the inner loop controls for the columns (months).

```
44              // compute average per region
45              for(int i = 0; i < NUM_REGIONS; ++i){
46                  double sum = 0;
47                  for(int j = 0; j < NUM_MONTHS; ++j)
48                      sum += sightings[i][j];
49                  System.out.printf("Average for Region %d:\t %.2f \n", i+1, sum / NUM_MONTHS);
50              }
51
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/flow_control$ javac BlackBears.java
kenneth@dragonborn:~/OneDrive/Spring2020/165/examples/flow_control$ java BlackBears black_bear_sightings.txt
Average for Region 1:    44.42
Average for Region 2:    51.42
Average for Region 3:    64.92
Average for Region 4:    33.08
Average for Region 5:    38.50
Average for Region 6:    47.75
Average for Region 7:    45.83
```