Control Flow Statements

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html
http://math.hws.edu/javanotes/c3/index.html

1 Control Flow

The basic building blocks of programs - variables, expressions, statements, etc. - can be put together to build complex programs with more interesting behavior. ***CONTROL FLOW STATEMENTS*** break up the flow of execution by employing decision making, looping, and branching, enabling your program to ***conditionally*** execute particular blocks of code. ***Decision-making statements*** include the `if` statements and `switch` statements. There are also ***looping statements***, as well as ***branching statements*** supported by Java.

2 Decision-Making Statements

A. `if` statement

```
if (x > 0)
     y++;   // execute this statement if the expression (x > 0) evaluates to "true"
            // if it doesn't evaluate to "true", this part is just skipped
            // and the code continues on with the subsequent lines
```

B. `if-else` statement -- gives another option if the expression by the `if` part evaluates to "`false`"

```
if (x > 0)
     y++;   // execute this statement if the expression (x > 0) evaluates to "true"
else
     z++;   // if expression doesn't evaluate to "true", then this part is executed instead
```

```
if (testScore >= 90)
     grade = 'A';
else if (testScore >= 80)
     grade = 'B';
else if (testScore >= 70)
     grade = 'C';
else if (testScore >= 60)
     grade = 'D';
else
     grade = 'F';
```

C. `switch` statement -- can be used in place of a big `if-then-else` statement; works with primitive types `byte`, `short`, `char`, and `int`; also with Strings, with Java SE7, (enclose the String with double quotes); as well as enumerated types,

```
int month = 8;
String monthString;
switch (month) {
       case 1:
              monthString = "January";
              break;
       case 2:
              monthString = "February";
              break;
       case 3:
              monthString = "March";
              break;

         etc ...

       default:
              monthString = "Invalid month";
              break;
}
System.out.println(monthString);
```

```
int monthNumber = 0;
switch (month) {
        case "January":
               monthNumber = 1;
               break;
        case "February":
               monthNumber = 2;
               break;

        etc …

        default:
               monthNumber = 0;
               break;
}
System.out.println(monthNumber);


enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }
Day day;
switch (day) {
        case MONDAY:
               System.out.println("Mondays are bad.");
               break;
        case FRIDAY:
               System.out.println("Fridays are better.");
               break;
        case SATURDAY:
        case SUNDAY:
               System.out.println("Weekends are best.");
               break;
        default:
               System.out.println("When will the weekend get here?");
               break;
}
```

3 Looping Statements

A. The `while` statement continually executes a block of statements while a particular condition is `true`. The `while` statement continues testing the expression and executing its block until the expression evaluates to `false`.

```
while (expression) {
       statement(s)
}
```

B. The `do-while` statement evaluates its expression at the bottom of the loop instead of the top, so the statements within the `do` block are guaranteed to execute at least once.

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

< - - notice the semi-colon at the end of the `do-while` statement - - every statement in Java ends in either a } or a ;

C. The `for` statement provides a way to iterate repeatedly until a particular condition is satisfied.

```
for (initialization; termination; increment/decrement) {
       statement(s)
}
```

There is another form of the `for` statement designed for iteration through arrays, sometimes referred to as the *enhanced for* statement, or `for-each`. In the following example, the variable `item` holds the current value from the `numbers` array.

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};
for (int item : numbers) {
       System.out.println("Count is: " + item);
}
```

4 Branching Statements

A. The `break` statement has two forms: labeled and unlabeled. The *unlabeled* `break` is like the one used in a `switch` statement. You can also used an unlabeled `break` to terminate a `for`, `while`, or `do-while` loop, although this practice is usually seen as sloppy programming and is discouraged by some.

```
for (i = 0; i < arrayOfInts.length; i++) {
     if (arrayOfInts[i] == searchfor) {
         foundIt = true;
         break;
     }
}
```

A *labeled* `break` statement terminates an outer statement that is labeled by some word. For example, if you have nested for loops, labeled with the word "search" right before the first for loop, you can put the following statement `break search;` inside the inner `for` loop to break out of both when the condition is met, and control flow continues with the statement immediately following the labeled statement.

```
search:
   for (i = 0; i < arrayOfInts.length; i++) {
       for (j = 0; j < arrayOfInts[i].length; j++) {
           if (arrayOfInts[i][j] == searchfor) {
               foundIt = true;
               break search;
           }
       }
    }
```

B. The `continue` statement skips the current iteration of a loop. The *unlabeled* form skips to the end of the innermost loop's body and evaluates the expression that controls the loop.

```
for (int i = 0; i < max; i++) {
     // interested only in p's
     if (searchMe.charAt(i) != 'p')
         continue;

     // process p's - only increments if it found a 'p'
     numPs++;
}
```

A *labeled* `continue` statement skips the current iteration of an outer loop marked with the given label.

```
test:
   for (int i = 0; i <= max; i++) {
       int n = substring.length();
       int j = i;
       int k = 0;
       while (n-- != 0) {
           if (searchMe.charAt(j++)
               != substring.charAt(k++)) {
               continue test;
           }
       }
       foundIt = true;
           break test;
    }
```

C. The `return` statement exits from the current method and returns control back to where the method was invoked from.