Java Control Flow

Ordinarily Java statements in a function or block are executed from the top down in the order they appear. However very rarely will statements simply flow from top to bottom with no interruption. Java has numerous ways to control the flow of the program. These can be grouped into three different types of statements:

- Decision statements
- Looping statements
- Branching statements

Each of these will be demonstrated and described in detail below.

Decision Statements

The simplest decision statement is the `if` statement. It simply is the keyword if followed by a boolean condition. If the condition is true the statement following the test is executed. If the statement is false the statement is not executed. Logically it looks like this:

```
if(condition)
  statement(s);
```

It would be of limited usefulness if only a single statement could be executed after the check. This is where statement blocks are really important. Statement blocks are delimited by the curly braces - `{` for the beginning, `}` for the end. So a more typical if statement would look like this:

```
int count = 44;
if ( count > 20)
{
    String msg = "Count is large!";
    System.out.println(msg);
}

// output:
// Count is large!
```

In this case the test is obvious since the variable was initialized right before the statement. But the important thing to remember is that if the expression in the parentheses is true the block executes, if it is false the block is skipped. To belabor that point the following code snippet shows a couple of `if` statements and the resulting output.

```java
int count = 44;
if ( count > 20 )
{
    String msg = "Count is large!";
    System.out.println(msg);
}

if ( count > 50 )
{
    String msg = "Count is VERY large!";
    System.out.println(msg);
}
System.out.println("Finished checking count.");

// output:
// Count is large!
// Finished checking count.
```

A note about formatting:
Much like variable names, formatting of `if` statements, and blocks in general, is not governed by the compiler. Convention, and more commonly a company coding standard, is what determines where braces are place, the number of spaces to indent, etc. Unlike some languages (such as Python) that relay on spaces for delimiting code blocks, Java uses the explicit brace delimiters `{ }` and where they are placed is not enforced by the compiler.

If statements can be nested. So the above code can be refactored to:

```java
int count = 44;
if ( count > 20 )
{
    String msg = "Count is large!";
    if ( count > 50 )
    {
        msg = msg + "\nCount is VERY large";
    }
    System.out.println(msg);
}
System.out.println("Finished checking count.");

// same output as above
```

One thing to keep in mind when using nested blocks is the variable scope rules that were discussed in the "Data Types and Variables" module. That is why the `msg` variable was used in the nested example and why it could be *reused* in the non-nested version.

So far the conditional check has been very simple, a single numeric comparison expression on a recently declared variable. However this need not be the case; any expression that results in a `boolean` output can be used and parentheses can be used for complex grouping. Frequently `if` statements are used to validate inputs to methods. The code snippet below shows this use case and a complex conditional expression.

```
void CheckId( boolean validId, int age, boolean buyingAlc )
{
    if (validId)
    {
        System.out.println("Valid ID presented");
        if((age>21) && (buyingAlc))
        {
            System.out.println("Customer legal to purchase alcohol");
        }
    }
}
```

Notice that the statements above only have limited usefulness because thus far only the postive outcome has been shown. So far it has been all or nothing - either the code inside the `if` block executes or it doesn't - there is no alternative path.

Thankfully the `if` statement does have an alternative path statement - the `else` block. This path is taken when the `if` condition is false. So to further expand and make the previous function validation much more useful the following code can be written:

```
void CheckId( boolean validId, int age, boolean buyingAlc )
{
    if (validId)
    {
        System.out.println("Valid ID presented");
        if((age>21) && (buyingAlc))
        {
            System.out.println("Customer legal to purchase alcohol");
        }
        else
        {
            System.out.println("No alcohol sales to this person");
        }
    }
    else
    {
        System.out.println("Error: Invalid license");
    }
}
```

To keep harping on the formatting issue for a minute, as a demonstration to show how different formatting can be, the above code is also commonly formatted as shown below:

```
void CheckId( boolean validId, int age, boolean buyingAlc ) {
    if (validId){
        System.out.println("Valid ID presented");
        if((age>21) && (buyingAlc)){
            System.out.println("Customer legal to purchase alcohol");
        } else {
            System.out.println("No alcohol sales to this person");
        }
    } else {
        System.out.println("Error: Invalid license");
    }
}
```

While more compact from a number-of-lines perspective, this type of formatting will not be used for the remaining code examples as it may be more difficult to discern the blocks, especially in the beginning. However it is important to recognize this formatting as it is prevalent in industry.

So with the `else` statement another path has been added. However, ignoring nesting there is basically a "this or that" decision path. But what is more is desired? A "this or that or the other" type of path? Java allows for this as well, in two different ways. The first is a simple extension of the `if else` statement into an `if else if` as the example below shows.

```
// imagine a List, someList, that is created previous to this code
int listSize = someList.size();

if (listSize > 100)
{
    System.out.println("List is really big!");
}
else if (listSize > 80)
{
    System.out.println("List is kind of big.");
}
else if (listSize > 60)
{
    System.out.println("List is just a little over half big.");
}
else if (listSize > 30 && listSize <= 59)
{
    System.out.println("List is approaching small")
}
else
```

```
{
    System.out.println("List is small");
}
```

The second way to decide between multiple potential valid paths is with a `switch` statement. When deciding which to use, one important consideration is the type of value to test against. While the `if-else` statement can evaluate complex boolean expressions, the `switch` statement only works with single values of `byte`, `short`, `char`, and `int` (or their wrapper classes); with instances of `String` objects, and enumerations (which will be discussed in a future module).
A `switch` statement puts the value to test against in parentheses, and a set of matches are listed in a block below, each one in the form of `case MATCH:`. When a match is found all the statements following that match will be executed until the statements end. The following example shows a way to produce an output based on t-shirt sizes:

```
public void printShirtSize(char shirt)
{
    String msg = "";
    switch(shirt)
    {
        case 's':
            msg = "Shirt is small";
            break;
        case 'm':
            msg = "Shirt is medium";
            break;
        case 'l':
            msg = "Shirt is large";
            break;
        case 'x':
            msg = "Shirt is Xtra larg";
            break;
        default:
            msg = "Unknown shirt size";
            break;
    }
    System.out.println(msg);
}
```

The value of `shirt` is tested against a set of known values. If a match is found the corresponding `String` will be placed in the `msg` variable. If no match is found the `default` block will be executed. Since `switch` works on a specific number of tests and not generic boolean expressions a default statement is critical to catching tests that do not match any values.

The `break` statement will be detailed later in this module, but the quick explanation is that when `break` is encountered, the `switch` ends and all remaining comparisons are skipped. This actually leads to a very convenient behavior when grouping choices with the same results. For example, the above statements assume the `shirt` parameter will be lowercase. But what happens if it is upper case? There are actually several different ways to check or force the case of the decision variable. But since the `case` statement is being described that is how it will be handled. The code below should print the correct shirt size regardless of the case of the `shirt` argument.

```
public void printShirtSize(char shirt)
{
    String msg = "";
    switch(shirt)
    {
        case 's':
        case 'S':
            msg = "Shirt is small";
            break;
        case 'm':
        case 'M':
            msg = "Shirt is medium";
            break;
        case 'l':
        case 'L':
            msg = "Shirt is large";
            break;
        case 'x':
        case 'X':
            msg = "Shirt is Xtra larg";
            break;
        default:
            msg = "Unknown shirt size";
            break;
    }
    System.out.println(msg);
}
```

Looping Statements

The next type of control statements to look at are the looping statements. These statements give the ability to process items in a repetitive manner. Think of processing items in a list, examining a String one character at a time, or reading a file a line at a time.

The most straight-forward loop to discuss is the `for` loop. This construct will execute its contained statements a specific number of times and is very flexible in terms of both its initialization and the way it flows through its process. Syntactically, the loop looks like this:

```
for(initial; end check; increment)
{
    statement(s);
}
```

- The `initial` statement is executed once as the loop begins and initializes the loop. It is only executed one time.
- After each iteration the `increment` statement is invoked. This statement can be positive or negative; linear or non-linear.
- The `end check` is evaluated after each increment; if it evaluates to `false` the loops quits.

The easiest way to demonstrate this loop is via a simple counter.

```
for(i=1; i<=20; i++)
{
    System.out.println("The current counter is: " + i);
}
// The output would be:
The current counter is: 1
The current counter is: 2
The current counter is: 3
The current counter is: 4
...
The current counter is: 20
```

The *iteration variable* `i` is only in scope for the duration of the loop, thus it is common to use a simple variable name. Also note that in the example above the simple post-increment operator is used. As statement above this is often used, but not a requirement. A loop to count down to 0 from a starting point would `i--`. A loop to print only odd or even numbers could use `i += 2`. In fact, all the statements of the for loop are optional so an intentional infinite loop can be created as follows:

```
for( ; ; )
{
    // statements.
}
```

Obviously this is not usually recommended as the code *inside* the loop has to have some way of ending its execution. The statements to do that will be discussed later in this module.

Collections and arrays will be discussed in a later module, but there is also an **enhanced** for loop that allows for easy processing of these data structures. These structures are known is *Iterable*, meaning they have some built-in plumbing to follow the Iterator pattern. In simple terms this means that the collection itself has the ability to produce each one of its elements in sequential order starting at the first element and going to the last element. To compare an array of integers will be

created, and then looped through with a normal `for` and an enhanced `for`.

```
int[] simpleArray = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// using the standard for loop and the length property of an array
for (int i = 0; i < simpleArray.length; i++)
{
    System.out.println("Normal for value: " + simpleArray[i]);
}

// using the enhanced for loop
for (int item : simpleArray)
{
    System.out.println("Enhanced for: " + item);
}

// both loops will print the their respective strings 1 - 10.
```

It is recommended to use enhanced `for` whenever possible.

The other looping structures used in Java are the `while` and `do-while` loops. These are essentially mirror images of each other, the only difference being when the end-check is evaluated. Life the `if` statement, the end-check can be a simple or complex boolean expression evaluation. Syntactically the loops look like this:

```
// while loop
while (expression)
{
    // do some stuff
}

// do-while loop
do
{
    // do some stuff
} while (expression)
```

The main difference in evaluating the end-check expression at different places is in the minimum number of times the loop will execute. If the expression is `false` the `while` loop will not execute, whereas the `do-while` loop will always execute at least once.

This simple example shows how an interactive console program could be built using a `do-while`.

```
char ch = 'c';

do
{
    // Do some processing, with or without value of ch

    System.out.print("Enter another char command or q to quit: ");

    ch = (char) System.in.read();

} while (ch != 'q');

System.out.println("Quit command entered!");
```

Another common use of the `while` loop is to processes resources that variable in size or length but have a defined way of signaling when there is no more data left to process. This could be arbitrary strings, database query resultsets, or data coming from disk files, or data returned from an internet call. The key is that the operation returns a predetermined value at the end. This could be the `null` value

```
BufferedReader reader = new BufferedReader(new FileReader("myTextFile.txt"));
String line = reader.readLine();
while (line != null)
{
    System.out.println(line);
    line = reader.readLine();
}
```

Depending on how the items are to be processed, the above statement could be further simplified by putting the check statement as part of the loop. In database processing, which will be discussed in a future module, the `ResultSet` class serves as the access point for both the next line of data and the individual items in that line. The `next()` call returns `false` if there are no more items to process.

```
// connection and query defined before this block - unimportant to while
statement.
Statement statement = connection.createStatement();
ResultSet rs = statement.executeQuery(query);
// rs is the result of the query
while (rs.next())
{
    String firstName = rs.getString("FIRST_NAME");
    String lastName = rs.getString("LAST_NAME");
    int empNum = rs.getFloat("EMPLOYEE_NUM");

    System.out.println(firstName + " " + LastName +
                       "\t" + empNum);
}
```

Note that unlike some languages, Java demands a `boolean` expression and there are no *boolean equivalent* values. Some languages allow testing integer values (0 is false, non-zero is true) or reference values (null is false, non-null is true) but Java does not. The full boolean expressions (val == 0) or (ref != null) must be used.

Keep in mind that unlike the `for` loops, the `while` and `do-while` must address the end-check variable withing the looping block. '

Here is a summary of the looping mechanisms and the caveats of each. | Loop | Syntax | Notes | |--------- |--------- | --------- | | for | `for (init; end-check; increment) {}` | All expressions are optional. Increment can be positive or negative. Loop ends when end-check is false.
| enhanced for | `for (var item: collection) {}` | Used for built-in collections that are iterable and custom collection that is iterable. | | while | `while (boolean expression) {}` | The boolean exression is checked first, if true the loop executes and the process repeats. | | do while | `do {} while(boolean expression)` | The loop executes and the boolean expression is checked. If true the process repeats. |

Branching Statements

The final type of control statement to examine is the branching statement. These have all have the common effect of immediately ending the execution of a block of code.

break
As was shown in the explanation of the `switch` statement, the `break` statement there is used to immediately end the execution of the `switch`. All statements after `break` are skipped and program execution resumes after the closing brace of the `switch`. The `break` statement can also appear inside of a loop with the same results - the loop is immediately exited and program execution resumes after the loop. This example shows how the break statement is used:

```
// generate a random number between 0 and 100
```

```
Random random = new Random();
int breakAt = random.nextInt(100);

// print a 'list' of random length
for(int i=0; i<100; i++)
{
    if ( i==breakAt)
    {
        System.out.println("Random number was: " + breakAt);
        break;
    }
    System.println("Index: " + i);
}
System.out.println("After the loop");
```

These statements above will print the numbers sequentially until the random number is reached. Then the `if` statement will evaluate to `true`, the "Random number was" statement will be printed, and the `for` loop will exit and the statement to print "After the loop" will be executed.
If there are nested loops, only the loop where the `break` statement is will be ended. If that is an inner loop, the outer loop will resume execution including re-executing the inner loop. So to extend the looping example above, if the goal was to create 5 different printouts of varying length, the following code could be used:

```
Random random = new Random();
int breakAt = random.nextInt(100);

for(int x=1; x<=5; x++)
{
    System.out.println("Starting list " + x);

    for(int i=0; i<100; i++)
    {
        if ( i==breakAt)
        {
            System.out.println("Length of this list: " + breakAt);
            break;
        }
        System.println("Index: " + i);
    }
    System.out.println("Finished list " + x);
    // Get the next randome number for a different size list
    breakAt = random.nextInt(100);
}
System.out.println("After the loop");
```

Breaking out of loops and `switch` statements are by far the most common form of the `break` statement. Technically however, this is known as the "unlabeled break" statement because it does not include an identifier or label. This statement will terminate the labeled statement regardless of where that is and execution will resume immediately after the labeled statement.
For a simple example assume that under some condition we wanted to stop generating lists altogether instead of simply moving on to the next list once the `breakAt` value is reached. Just for the sake of example let's say that if the `breakAt` value is divided by the number of the list (`x`) with a remainder of 2, we want to stop generating lists altogether. For concrete reference, that would happen if the `breakAt` value was 32 and list number 3 was being processed - 32 divided by 3 is 10 with a remainder of 2. So instead of all 5 lists being generated only 3 would be.

```
Random random = new Random();
int breakAt = random.nextInt(100);

outer:
for(int x=1; x<=5; x++)
{
    System.out.println("Starting list " + x);

    for(int i=0; i<100; i++)
    {
        if ( i==breakAt)
        {
            System.out.println("Length of this list: " + breakAt);

            if (breakAt%x == 2)
            {
                System.out.println("List generation ended early!");
                break outer;
            }
            else
            {
                break;
            }
        }
        System.println("Index: " + i);
    }
    System.out.println("Finished list " + x);
    // Get the next randome number for a different size list
    breakAt = random.nextInt(100);
}
System.out.println("After the loop");
```

When the statement `breakAt%x == 2` is true the `break outer` statement causes the program to immediately end the statement after the label, which is the outer `for x<=5` loop. If this is the case the final two statements printed will be:

```
List generation ended early!
After the loop
```

continue

The `continue` statement is similar but complimentary to the `break` statement. It can appear in the same looping statements, but rather than exiting the loop altogether it will simply skip any statements after it and resume the next iteration of the loop. To further operate on the original list example from above, assume that everything is the same except now only odd numbers will be printed. The code to do that is:

```
// generate a random number between 0 and 100
Random random = new Random();
int breakAt = random.nextInt(100);

// print a 'list' of random length
for(int i=0; i<100; i++)
{
    if ( i==breakAt)
    {
        System.out.println("Random number was: " + breakAt);
        break;
    }
    // if number is even, skip to next iteration
    if ( i%2==0)
    {
        continue;
    }
    System.println("Index: " + i);
}
System.out.println("After the loop");
```

Notice that `break` and `continue` can exist in the same loop but have different checks and will have different effects. The `break` statement completely exits the loop, potentially skipping over some of the chosen indexes, while the `continue` statement will allow the loop to execute for each index but may skip some of the statements.

As with `break`, when `continue` is in a nested loop it will only skip the remaining statements within its own loop, and the loop will pick up at the next iteration. There is also a labeled form of `continue` which will skip the remainder of the labeled loop's statements and resume with the next iteration.

```
// the snippet would print 5 'lists', each with the numbers 0-12 in them.
```

```
outer:
for(int i=0; i<5; i++)
{
    System.out.println("List number: " + i);
    for(int y=0; y<25; y++)
    {
        System.out.println("Index: " + y);
        if ( y == 12)
        {
            continue outer;
        }
    }
}
```

return

The final branching statement is `return`, which is used to exit a method. This is considered a branching statement because it can appear anywhere within a method. There are some best practices and conventions regarding whether or not multiple return statements should be present in a function, but the compiler doesn't enforce any rules regarding this. From the compiler's standpoint a `return` statement can appear anywhere in a method and when encountered it will cause an immediate end to the method. \

There are two forms of `return`, one with just the plain statement and one with a statement and a value or identifier. The compiler *will* enforce that the `return` statement provide a value of the appropriate type if the method has a return value, or that the value return nothing if the method is of type `void`. The following examples show both types of return statements, and also demonstrate the two design methods for using `return`; one with multiple return paths and another with a single `return` path.

```
 // insert ONLY positive, odd data
public void insertOddData(int data)
{
    // equal to 0 check
    if (data == 0)
    {
        return;
    }
    // Negative check
    if (data < 0)
    {
        return;
    }
    // check to see if data is even or odd
    if ( data%2 == 0)
    {
```

```
            return;
        }

        // some code here to store the odd data somewhere
        System.out.println("Odd Data Stored")
        return;
    }

    // find and return the sum of the array
    public int getArraySum(int[] arr)
    {
        // initialize to a valid default return value
        int sum = 0;

        // array must have values to sum
        if (arr.length > 0)
        {
            for( int n : arr)
            {
                sum += n;
            }
        }

        return sum;
    }
```

--

--

--

--

Questions

Which looping type(s) will always execute at least once?

- For
- Enhanced For
- While
- Do-While

Which loop works best with built-in collection types?

- For
- Enhanced For
- While

- Do-While

Which decision statement(s) can have complex boolean expressions to evaluate?

- if
- switch

A return statement must always include a value return

- True
- False

Which control statement will immediately end a loop and resume execution after the loop?

- break
- continue
- return

If you want to make sure you iterate through each iteration of a loop but skip some statements based on some condition you should use:

- break
- continue
- return
- none of the above

Which looping statement can be used to create an infinite loop?

- for
- while
- neither
- both

Muliple conditions can best be met in a switch statement by

- using complex boolean expressions to match against
- combining multiple case sections to have a common statement block
- repeating the statement block in each case that matches
- repeating the switch statment with the different conditions that match