Learning Notes for Unit 2 – Part A

Comparison Operators:

Boolean logic is based on the comparison of two "items" that results in a **true** or **false** answer. You would use this in a program if you need to check and see if a certain condition is met and then your program responds differently depending on the answer.

For example, let's assume you are storing a bank account balance and you want to display a message if your account goes into overdraft. You would have a variable that stores your current balance and you would compare it to the number 0. If the balance is less than 0, you display a message. If the balance is 0 or more, you would not.

There are number of comparison operators that you can use in your programs.:

- **<** Compares the value of the left-hand-side of the operator (**LHS**) to the right-hand-side (**RHS**) of the operator. Result is **true** if LHS is less than RHS. Result is **false** if LHS is greater than or equal to RHS

- **<=** Result is **true** if LHS is less than or equal to RHS. Result is **false** if LHS is greater than RHS

- **>** Result is **true** if LHS is greater than RHS. Result is **false** if LHS is less than or equal to RHS

- **>=** Result is **true** if LHS is greater than or equal to RHS. Result is **false** if LHS is less than RHS

- **==** Result is **true** if LHS is equal to RHS. Result is **false** if LHS is not equal to RHS

- **!=** Result is **true** if LHS is not equal to RHS. Result is **false** if LHS is equal to RHS

Comparing Strings:

If you are wanting to compare two String variables, you should not use the **==** notation. You should use the .equals("string to compare to") notation to do the comparison.

Example:

String TestString = "Hello";

if (TestString == "Hello") … <= Should not use this method (not valid for early Java releases)

if (TestString.equals("Hello")) … <= Use This Method (Accepted Approach)

Multiple Comparisons - Boolean Operators:

Sometimes comparing two values isn't enough information to make a decision. For example, you a common case is when you may encounter division by zero.

Let's say you want to perform a block of code when the ratio of x to y is less than 1. Your test would look like:

if (x/y < 1) …

If **y is 0**, **your program will crash** because division by zero (#/0) does not produce a result your program can handle.

To prevent this from happening, you need to verify that x/y < 0 AND y!=0

You could rewrite the test as follows:

if (x/y < 1 && y!=0) …

This is better, but your program could still crash as the x/y< 1 comparison is done first.

If you rewrite the test as :

if (y!=0 && x/y < 1) …

Then your **program will not crash** if y is equal to 0. Why? Short Circuit Evaluation (see below).

We have introduced two concepts in this section:

1. Comparing the results of individual tests to get an overall true/false.
2. **Short Circuit Evaluation** – As soon as the compiler knows what the overall outcome of the test will be, the testing stops and the program proceeds

Concept 1: Comparing Multiple Tests

To compare multiple tests to get an overall result, the two Boolean test operators that are used are:

&& - AND comparison operator. It compares the Boolean result (from a comparison test) on the LHS to the Boolean result (also from a comparison test) on the RHS. If the **LHS** AND the **RHS** are **BOTH TRUE**, this test returns **true**.

|| - OR comparison operator (two pipe characters – found above enter key)
If **either the LHS or the RHS are true**, the test returns **true**

Note: The AND test (&&) only returns true when **both** the **LHS AND RHS** are true. A common mistake to make is to think that it returns true when the LHS AND RHS are the same. This is incorrect. **If LHS AND RHS tests are both false, it returns FALSE**.

Concept 2: Short Circuit Evaluation

Because of the nature of the AND/OR comparisons, some shortcuts can be used by the interpreter.

If you are comparing multiple tests with only ANDs (&&), then you can stop the evaluation as soon as the first 'false' test is seen because no matter what follows, the test will never be TRUE.

For example, in the division by zero code from above,

if (y!=0 && x/y < 1) …

as soon as the first test is false (**when y IS 0**), then **the testing stops** because no matter what the result of x/y < 1 is, the overall test result will never be true because of the AND (&&) operator

Likewise, when you are comparing multiple tests with only ORs (||), the testing can stop as soon as the first true is found. As soon as the first TRUE is encountered, nothing that follows can make the overall test FALSE

IF statements:

As shown earlier, if statements are used when you want to perform a statement (or series of statements) when a result of a comparison test is found to be true.

Syntax:

```
if ( test to be evaluated ) {

    //code to be executed if the test is true

}
```

Control statements (introduced in this unit) do not have a semicolon (;) at the end of each line. Instead, curly braces are used to indicate where a block of code associated with the statement starts and where it ends.

Each statement inside the if code block MUST end with a semicolon as you would expect.

IF/ELSE statements:

This control statement has the same general function as the IF statement described above, but in this case, you WILL do an alternate block of code rather than doing nothing and proceeding with the program.

Syntax:

```
if ( test to be evaluated ) {
    //code to be executed if the test is true
} else {
    //code to be executed if the test is false
}
```

Nesting IF statements:

Sometimes your pseudocode may have extra comparison tests to perform inside an if test block.

Taking the bank account example mentioned above, let's modify it to display an "empty" message if the balance is 0, a "too low" message if the balance is between 0 and 50, and a "low message" if the balance is between 50 and 100.

The pseudocode would look like this:

```
// if the account balance is less than 0
        //display overdraft message
// otherwise
        //if account balance is 0
                //display empty
        //otherwise
                //if account balance is less than 50
                        //display too low
                //otherwise
                        //if account balance is less than 100
                                //display low
```

Please note two things with the above pseudocode:

- The indenting – Every time you evaluate a condition and do an action, indent. Your code is so much more readable.
- The pseudocode itself tells you what you need to write. You will NOT get tripped up with the syntax if you have a proper plan

Your implemented code will look like this:

```
// if the account balance is less than 0
if (balance < 0 ) {

        //display overdraft message
        System.out.println("Overdraft");

// otherwise
} else {

        //if account balance is 0
        if (balance == 0 ) {

                //display empty
                System.out.println("Empty");

        //otherwise
        } else {

                //if account balance is less than 50
                if (balance < 50 ) {

                        //display too low
                        System.out.println("Too Low");
```

```
            //otherwise
            } else {

                //if account balance is less than 100
                if (balance < 100 ) {

                    //display low
                    System.out.println("Low");

                }
            }
        }
}
```

Notice how the closing brackets match the indentation level

You will notice that it can get messy and confusion when you nest if statements too deeply. Many sources recommend that you do not nest more than 3 levels deep.

Alternately you could have written the code as follows:

```
// if the account balance is less than 0
        //display overdraft message

// if the account balance is 0
        //display empty

//if account balance is between 0 and 50
        //display too low

//if account balance is between 50 and 100
        //display low
```

With Java code:

```
// if the account balance is less than 0
if (balance < 0 ) {

        //display overdraft message
        System.out.println("Overdraft");

}
// if the account balance is 0
if (balance == 0 ) {

        //display empty
        System.out.println("Empty");

}
//if account balance is between 0 and 50
if (balance > 0 && balance <50 ) {

        //display too low
        System.out.println("Too Low");

}
//if account balance is between 50 and 100
if (balance >= 50 && balance <100 ) {

        //display low
        System.out.println("Low");

}
```

You get “cleaner” code, but there is a tradeoff. Each one of these test will run no matter what, requiring more CPU cycles. If you are working in an environment with limited system resources, this can cause problems. The nested approach is optimized as you get out of the overall test as soon as a true condition is evaluated and the rest of the tests are skipped.

SWITCH statements

The switch statement can be used when you have a series of if conditions that are discrete in nature (i.e. comparison against a single integer).

```
switch (balance) {

        case 1: statement(s); break;    //statement(s) done if balance is 1
        case 2: statement(s); break;    //statement(s) done if balance is 2
        case 3: statement(s); break;    //statement(s) done if balance is 3
        case 4: statement(s); break;    //statement(s) done if balance is 4
        .
        .
        .
        default: statement(s);          //statement(s) done if no matches

}
```

An example of use would be if you had an array of numbers (1-52) which represent a deck of cards, you would use a switch when you retrieve a number from the array and want to see what card it is.

```
switch (cardNumber) {

        case 1: picture=”hearts_ace.jpg”; value=1; break;
        case 2: picture=”hearts_two.jpg”; value=2; break;
        case 3: picture=”hearts_three.jpg”; value=3; break;
        case 4: picture=”hearts_four.jpg”; value=4; break;
        .
        .
        .
        default: System.out.println(“No Match”);

}
```

The switch statement would look at the value of cardNumber, scan the cases to find a match, and then execute the statements following the matching case. If there is no match, the default code will be executed.

As of the Java7 API release, you can now use **String**s or **char**s in the switch statement.

```
switch (month) {

        case "january":  monthNumber = 1;  break;
        case "february": monthNumber = 2; break;
        case "march": monthNumber = 3; break;
         …

        default: System.out.println(“No Match”);
}
```

Learning Notes for Unit 2 – Part B

A loop is a series of statements in a programming block that is executed repeatedly until a test condition becomes false.

There are two types of loop conditionals that you can use:

- The WHILE loop – is done if you do not know the precise number of times a block of code will be executed. If your pseudocode has a statement like "repeat until" then you will likely use a while loop

- The FOR loop – is done if you do know how many times a block of code will be executed. Loop through the array and add the values would be an example of a for loop since an array has a pre-determined size.

WHILE loops:

The while loop has the following syntax:

```
while (test) {

        //code to execute

}
```

At the start of each loop cycle, the test is evaluated to determine if it is true. If it is, the code in the block will execute and the test will be evaluated again at the start of the next cycle.

You would want this test to be true as you initially encounter this block of code (otherwise it will never run). The statements within the block would execute repeatedly until something happens to make your test false. If the test never becomes false, your have an **infinite loop** and your program will never proceed past that point.

Usually, an event happens that makes your test false, you would finish the sequence of statements and when the test is evaluated at the start of the next cycle, you would leave the loop and proceed with your program.

If you need to leave the loop immediately, you can use the **break;** statement to immediately leave the block of code.

Note: There is a variation of the while loop called the do…while loop. The test is evaluated at the END of each cycle which means the loop block is guaranteed to run at least one time.

FOR loops:

The for loop has the following syntax:

```
for ( counter=initial_value;  test;  counter_step ) {

        //code to execute

}
```

At the start of the loop, the counter is set to an initial value. This is **only done** when the loop is encountered for the **first time**.

At the start of each loop cycle, the test is evaluated to determine if it is still true. If it is, the code in the block will be executed.

At the end of each loop cycle, the counter value will change and then the test is evaluated again as the next loop cycle starts.

For example:

```
for ( int i=0;  i<10;  i++ ) {

        //code to execute

}
```

At the beginning of the loop, a counter variable, **i**, is created and initialized to 0. The loop runs as long as **i** is less than 10. At the end of each cycle, the value of **i** is increased by 1.

As a result, this loop will run 10 times (**i** values from 0 to 9 inclusive). As soon as i becomes 10, the test is false and the loop is abandoned.

Shortcut operators:

As you will observe from the for example, you will see **i++** as the counter step. This is a shorthand form for **i=i+1;**

Accepted shorthand operators

i**++;** is equivalent to **i=i+1;**

i--; is equivalent to **i=i-1;**

i+=2; is equivalent to **i=i+2;**

i-=2; is equivalent to **i=i-2;**

i*=2; is equivalent to **i=i*2;**

i/=2; is equivalent to **i=i/2;**

NESTED FOR/WHILE loops:

As demonstrated for IF loops, you can nest FOR or WHILE loops within each other if your solution requires it.

```
while (test1) {
        while (test2) {
        }
}
```

```
for (int i=0; i<10; i++) {
        for (int j=0; j<10; j++) {
        }
}
```

Because of the complexity, be careful when nesting control structures and only do it when absolutely necessary for your program.

Note: For loops are commonly used to access sequential values in an array. Nested for loops are commonly used to access sequential values in multidimensional arrays.