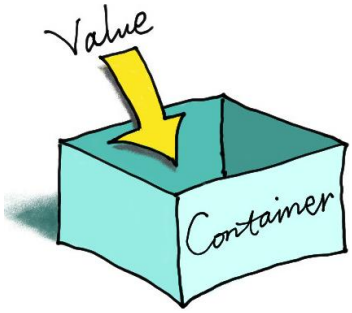
Constants, expressions, statements, procedures, and libraries

Variables are containers that be assigned values. These values can be numerical, alphabetically, symbolic, or any other data type. Variables can be declared without an assigned value, declared as so:

```
String myName;
```



But we want to assign a value to make this variable more useful. Once we assign a value to the variable, known as "Initialization", we can call the variable to use the value throughout the program. Below we will assign a name to the variable:

```
String myName = "George";
```

The variable (container) is myName and the value is "George". Variables can change value during program execution, but constants cannot change their value.

Constants

Constants is an identifier with an associated value that cannot be altered by the program. Hence, the value is constant. For example, in mathematics, the symbol:

$\pi = 3.1415927\ldots.$

As you know, the value for this symbol cannot change. Constants in programming can be user-declared, but after doing so, the program cannot change these constant values. Below is a few examples of programming constants.

```
const int height = 100;                    // Constants can be integers,
const float number = 4.32;                 // numbers with decimals,
const char letter = 'D';                   // letters,
const char letter_sequence[10] = "ABC";    // strings,
const char question_mark = '\?';           // and even symbols
```

Above are a few examples of constants in C programming language. The format will vary slightly to different programming languages. The concept would be the same: once declared the constant cannot be changed and can be called by its name (e.g. height, number, question_mark, etc.) for the assigned value.

When is this helpful? Birthdate would be an excellent example. The birthday of an individual would not change and the age can be calculated.

Statements

Statements tell the computer to do an action. They are similar to imperatives in English. (e.g. Go to sleep! Take the cannoli! Walk the dog! Etc.) There are several kinds of statements:

- if statements
- if…else statements
- Nested if…else statement (if…elseif…else statement)
- for loops
- do…while loops
- break and continue
- switch case statement
- goto statement

Statements will be covered during the programming portion of the TACT workshop, but review of material beforehand is encouraged.

if statement

The if-statement evaluate an expression inside parenthesis. If the expression is true, then the statements inside the if-statement are executed.

```
if (testExpression) {
        //statements
}
```

Of course, this isn't very helpful without seeing how it can be applied. Let's say a program wants to print "Congratulations!" if the user got a job promotion:

```
if (jobPromotion == true) {
        Print ("Congratulations!")
}
```

As you can see the expression in this case is:

jobPromotion == true

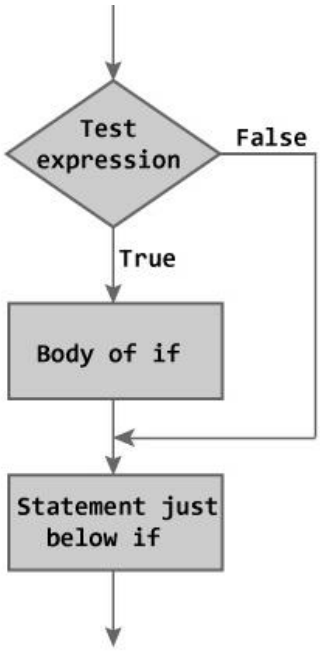


Figure: Flowchart of if Statement

If the statement is true, the statements within the if statement will execute. Another way to write the same expression is:

```
if (jobPromotion) {
        Print ("Congratulations!")
}
```

If there is no explicit comparison in the expression, the if-statement will see if the expression is true. If statements are useful to execute code for particular situations. You would not congratulate everybody, only those who got a job promotion. If the if-statement is false, the program will ignore the if-statement and continue right after/below the if-statement.

if…else statements

The if-else statements will execute the code if the expression is true, but if the expression is false then the statements within the else-statement are executed. Let's use the previous example, but this time, you want to print to the screen a message for every user running the program.

```
if (jobPromotion) {
        Print ("Congratulations!")
}
else  {
        Print ("Better luck next time.")
}
// Continue code here
```

Now, if there were several applicants to the job promotion, every person would get one of the two messages above. As before, if the user got the job promotion, then the program will print "Congratulations!" and then continue the code right after/below the if-else statement.

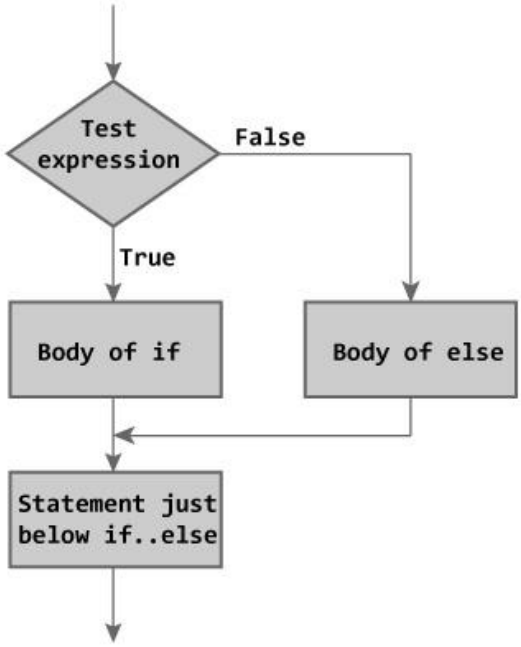


Figure: Flowchart of if...else Statement

If an applicant who did not get the promotion executed this program, then the following will happen.

1. The program will see if jobPromotion is true
2. jobPromotion is false, so move onto the next statement
3. Since the if-expression is false, we will execute the else-statement
4. The program will print "Better luck next time."
5. Program will continue code below the if-else statement.

Nested if…else statements (if…elseif…else statements)

Now we are going to apply additional expressions to evaluate after one another.

```
if (testExpression1) {
  // statements to be executed if testExpression1 is true
}
else if(testExpression2) {
  // statements to be executed if testExpression1 is false and testExpression2 is true
}
else if (testExpression 3) {
  // statements to be executed if testExpression1 and testExpression2 is false and testExpression3 is true
}
else {
  // statements to be executed if all test expressions are false
}
```

So the program will test each if-statement until it has a true statement. If the there is a true statement, then the statements included will execute and the program will continue right after/below the if-else statement. If all if-statements are false, then the else-statement is executed and the program will continue afterwards. Let's do an example with the direction the user if facing:

```
if (direction == "North") {
        print ("You are facing North")
}
elseif (direction == "East") {
        print ("You are facing East")
}
elseif (direction == "West") {
        print ("You are facing West")
}
else {
        print ("You are facing South")
}
```

Notice how I do not test to see if the direction is south. I did not include an elseif-statement for south because after testing the other three directions, the only one left is south. Having ruled out all other directions, placing the remaining statement in the else-statement is fair. With elseif-statements, there is no limit on how many that can be added.

Of course the code above is not the best way to write such an example. How would you have done it?

```
print ("You are facing "+ direction)
```

This is a much faster, cleaner line of code that would do the same thing as the if-else statement above. This line uses the value of variable "direction" to be included in your print statement.

For loops

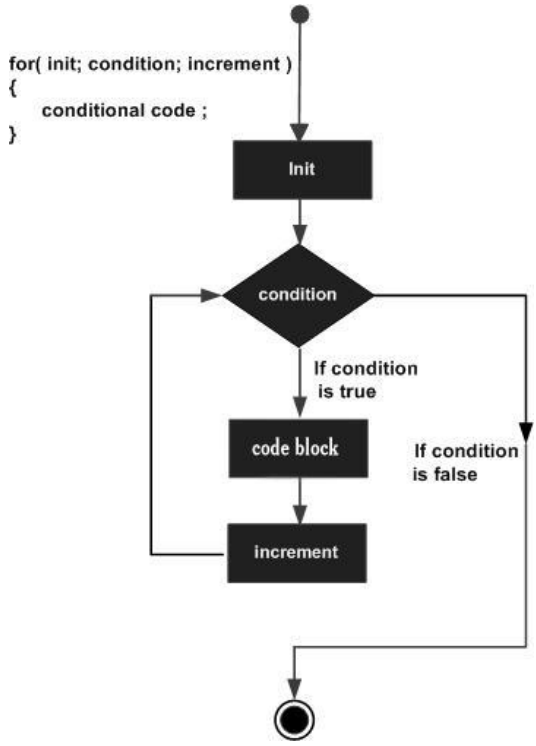
The for-loops are great for repetitive execution of a set of statements. A for loop consists of four parameters: init, test, update, and statements. Although this will vary slightly with some programming languages, but this is the syntax for C/C++ and Java.

```
for (init; test; update) {
        statements
}
```

Init : Statement executed when beginning loop
Test : If the test evaluations to true, the statements execute
Update : Executes at the end of the iteration
Statements: Collection of statements executed every time through the loop

```
for ( int j ← 1;  j  <  4;  j ← j + 1) {
        print ( “Number of messages printed: “ + j )
}
```

Init : int j ← 1
Test : j < 4
Update : j ← j + 1
Statements: print (“Number of messages printed: “ + j)



When the for loop executes, the following events occur:

1. The init statement is run.
2. The test is evaluated to be true or false.
3. If the test is true, jump to step 4. If the test is false, jump to step 6.
4. Run the statements within the block.
5. Run the update statement and jump to step 2.
6. Exit the loop.

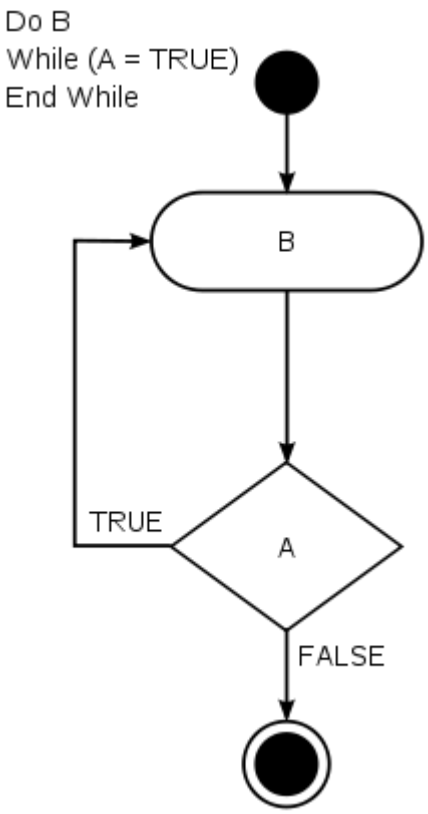
So you can think of the init as a sort of counter. This variable can start on any number but commonly started at 0 or 1. The update is adding 1 to the init variable after every loop. So when the test comes around there are four tests performed and three messages printed.

1 < 4 print – Number of messages printed: 1
2 < 4 print – Number of messages printed: 2
3 < 4 print – Number of messages printed: 3
4 < 4 4 is not less than 4 so we exit the for loop without executing the statements.

The for-loop can hold multiple statements, including more for-loops. If there is a nested for loop, for each iteration, the for-loop on the inside will start and complete before exiting to the first for-loop.

```
for (inti; test; update) {
        // Statement executes
        for (init; test; update) {
                // This loop will start and finish every time
        }
        // Statements here will execute after the second for loop
}
```

Do…while loops



Do-while loops are another iteration statement. Do-while loops will execute a block of code and check the expression to see if it is true. If so, the code block is repeated. Otherwise, the program continues. So in short, do-while looks are great for code that ensures the code will execute at least once.

```
do {
        // Statements
} while ( condition );
```

A real world example of a do-while loop is like a bus. There are multiple stops throughout the city that the bus goes through. You get on the bus to get to your destination so you need to sit on the bus at least once, but you may have to stay for a couple stops until you get to your destination.

```
do {
        rideBus();
} while (destinationNotReached);
```

So if you make it to your destination in one stop, then the while condition is not true and exits the do-while loop. Otherwise, you would continue to ride the bus until your destination.

Break and continue

Break will break a loop statement and continue executing the code after the loop (if any).

```
for (int k ← 0; k < 3; k ← k + 1) {
        if (k == 1) {   break;   }
        print (“This will print once and break the loop on the second iteration”)
}
// Continue executing code
```

Continue breaks one iteration (in a loop), and continues with the next iteration in the loop.

```
for (int k ← 1; k < 4; k ← k + 1) {
        if (k == 2) {   continue;   }
        print (“This will print, skip the second print, and print a third time”)
}
// Continue executing code
```

Switch case statement

Switch case statement performs different actions based on different conditions. An expression is compared to case statements and the corresponding code is executed. This may feel familiar to the if-elseif-else statement. For example, we can look at a “day” variable in a switch case statement:

```
switch (day)
 {
  case 1 : cout << "Sunday";
        break;
  case 2 : cout << "Monday";
        break;
  case 3 : cout << "Tuesday";
        break;
  case 4 : cout << "Wednesday";
        break;
  case 5 : cout << "Thursday";
        break;
  case 6 : cout << "Friday";
        break;
  case 7 : cout << "Saturday";
        break;
 default : cout << "Not an allowable day number";
        break;
 }
```

So this example looks at a “day” variable. If the variable has an assigned value of 1, then the output will be “Sunday”. If the value is 2, then the output will be “Monday”, and so on. The default case is in the event that there are no case statements that match the variable. Cases can also be combined in a switch case statement as so [syntax will vary slightly with different programming languages]:

```
switch (day)
 {
  case 1,6,7 : cout << "It’s the weekend!";
        break;
  case 2,3,4,5 : cout << "It’s a weekday.";
        break;
 default : cout << "Not an allowable day number";
        break;
 }
```

goto statement

A goto statement is a one-way transfer to another line of code. It tells the program to jump to another line of code. The goto statement is not available in every programming language. Below is an example:

```
print (“Hello world!”)
goto line 4
print (“This will be jumped and not printed”)
print (“Hello from line 4!”)
```

Goto statements can jump multiple lines and even jump back up the code. goto is a reserved word in Java and not used.

Expressions

Expressions consist of at least one **operand** (objects that are manipulated) and zero or more operators. Here are some examples:

```
52
32 + x                    [x and 32 are operands and + is the operator]
c = a + b
area = pi * radius * radius
```

An expression can be a statement but a statement is not always an expression.

Procedure

A **procedure** is a section of a program that performs a specific task. This sounds similar to a function but there is a difference. A function returns a value, while a procedure just executes commands. The difference is only that a function returns a value.

Libraries

A **library** is a collection of precompiled routines that a program can use. For instance, you want to write a program that reverses the letters of a given word, such as:

“Reverse” → “esreveR”

This could take some time and several lines of code, but with a library, you could call a method to do this instantly. In java, the method is called “.reverse()”. Once this method is called to the given string, the result will be a string with the letters reversed. Libraries are useful in the reuse in behavior and ease of distribution of code. Libraries consists of many various behaviors and methods.