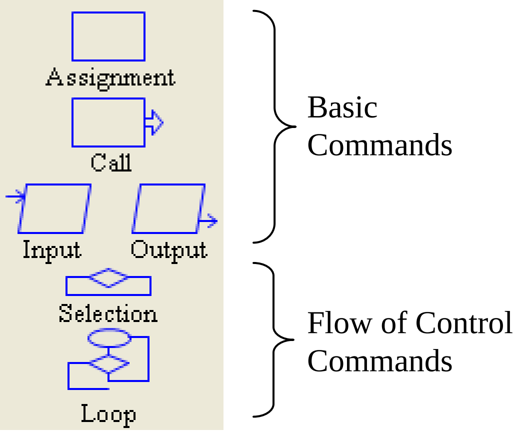
Programming Control Structures

By Dr. Wayne Brown

Introduction

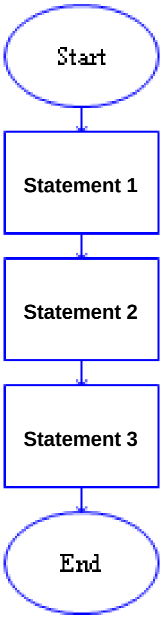
One of the most important aspects of programming is controlling which statement will execute next. *Control structures* / *Control statements* enable a programmer to determine the order in which program statements are executed. These control structures allow you to do two things: 1) skip some statements while executing others, and 2) repeat one or more statements while some condition is true.

RAPTOR programs use six basic types of statements, as shown in the figure to the right. You have already learned about the four basic commands in a previous reading. In this reading you will learn about the **`Selection`** and **`Loop`** commands.



Sequential Program Control

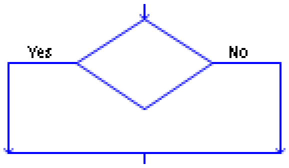
All of the RAPTOR programs you have seen in previous readings have used *sequential program control*. By sequential we mean "in sequence," one-after-the-other. Sequential logic is the easiest to construct and follow. Essentially you place each statement in the order that you want them to be executed and the program executes them in sequence from the `Start` statement to the `End` statement. As you can see by the example program to the right, the arrows linking the statements depict the execution flow. If your program included 20 basic commands then it would execute those 20 statements in order and then quit.



When you are solving a problem as a programmer, you must determine what statements are needed to create a solution to the problem and the order in which those statements must be executed. Writing the correct statements is one task. Determining where to place those statements in your program is equally important. For example, when you want to get and process data from the user you have to `GET` the data before you can use it. Switching the order of these statements would produce an invalid program.

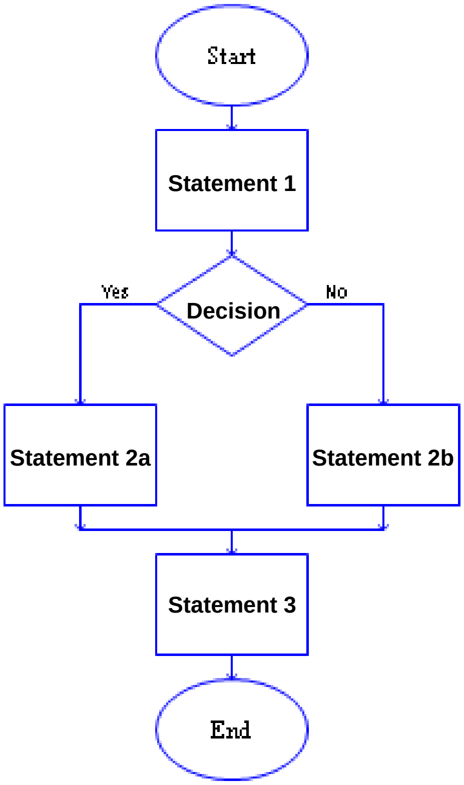
Sequential control is the "default" control in the sense that every statement automatically points to the next statement in the flowchart diagram. You do not need to do any extra work to make sequential control happen. However, using sequential control alone will not allow the development of solutions for most real-world problems. Most real world problems include "conditions" that determine what should be done next. For example, "If it is after taps, then turn your lights out," requires a decision to be made based on the time of day. The "condition" (i.e., the current time of day) determines whether the action should be executed or not executed. This is called "selection control" and is introduced next.

Selection Control



It is common that you will need to make a decision about some condition of your program's data to determine whether certain statements should be executed. For example, if you were calculating the slope of a line using the assignment statement, `slope ← dy / dx`, then you need to make sure that the value of `dx` is not zero (because division by zero is mathematically undefined and will produce a run-time error). Therefore, the decision you would need to make is, "Is `dx` zero?"

A *selection-control statement* allows you to make "decisions" in your code about the current state of your program's data and then to take one of two alternative paths to a "next" statement. The RAPTOR code on the right illustrates a selection-control statement, which is always drawn as a diamond. All decisions are stated as "yes/no" questions. When a program is executed, if the answer to a decision is "yes" (or true), then the left branch of control is taken. If the answer is "no" (or false), then the right branch of control is taken. In the example to the right, either statement 2a or statement 2b will be executed, but never both. Note that there are two possible executions of this example program:



Possibility 1	Possibility 2
Statement 1 Statement 2a Statement 3	Statement 1 Statement 2b Statement 3

Also note that either of the two paths of a selection-control statement could be empty or could contain several statements. It would be inappropriate for both paths to be empty or for both paths to have the exact same statements, because then your decision, Yes or No, would have no effect during program execution (since nothing different would happen based on the decision).

Decision Expressions

A selection-control statement requires an expression that can be evaluated into a "Yes/No" (or True/False) value. A decision expression is a combination of values (either constants or variables) and operators. Please carefully study the following rules for constructing valid decision expressions.

As you hopefully recall from our discussion of assignment statement expressions, a computer can only perform one operation at a time. When a decision expression is evaluated, the operations of the expression are not evaluated from left to right in the order that you typed them. Rather, the operations are performed based on a predefined "order of precedence." The order that operations are performed can make a radical difference in the final "Yes/No" value that is computed. You can always explicitly control the order in which operations are performed by grouping values and operators in parenthesis. Since decision expressions can contain calculations similar to those found in assignment statements, the following "order of precedence" must include assignment statement expression operators. The "order of precedence" for evaluating decision expression is:

1. compute all functions, then
2. compute anything in parentheses, then
3. compute exponentiation (^,**) i.e., raise one number to a power, then
4. compute multiplications and divisions, left to right, then
5. compute additions and subtractions, left to right, then
6. evaluate relational operators (= `!=` `/=` `<` `<=` `>` `>=`), left to right,
7. evaluate any **`not`** logical operators, left to right,
8. evaluate any **`and`** logical operators, left to right,
9. evaluate any **`xor`** logical operators, left to right, then finally
10. evaluate any **`or`** logical operators, left to right.

In the above list, the *relational* and *logical* operators are new. These new operators are explained in the following table.

Operation	**Description**	**Example**
=	"is equal to"	`3 = 4` is `No(false)`

`!=`
`/=`
	"is not equal to"	`3 != 4` is `Yes(true)`
		`3 /= 4` is `Yes(true)`

<	"is less than"	`3 < 4` is `Yes(true)`
<=	"is less than or equal to"	`3 <= 4` is `Yes(true)`
>	"is greater than"	`3 > 4` is `No(false)`
>=	"is greater than or equal to"	`3 >= 4` is `No(false)`
`and`	`Yes(true)` if **both** are `Yes`	`(3 < 4) and (10 < 20) is Yes(true)`
`or`	`Yes(true)` if **either** are `Yes`	`(3 < 4) or (10 > 20) is Yes(true)`
`xor`	`Yes(true)` if the "yes/no" values are not equal	`Yes xor No` is `Yes(true)`
`not`	Invert the logic of the value `Yes` if `No`; `No` if `Yes`	`not (3 < 4)` is `No(false)`

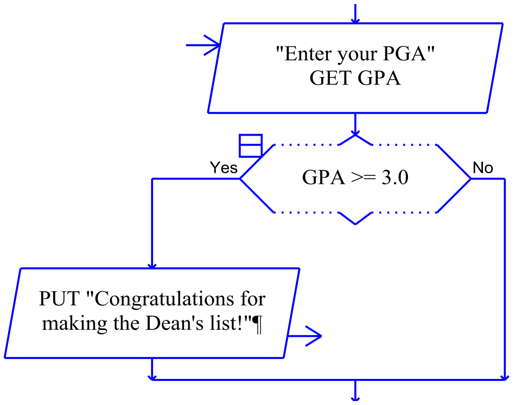
The relational operators, (= `!=` `/=` `<` `<=` `>` `>=`), must always compare two values of the same data type (either numbers, text, or "yes/no" values). For example, `3 = 4` or `"Wayne" = "Sam"` are valid comparisons, but `3 = "Mike"` is invalid.

The logical operators, (`and` , `or`, `xor`), must always combine two Boolean values (true/false) into a single Boolean value. The logical operator, `not`, must always convert a single Boolean value into its opposite truth value. Several valid and invalid examples of decision expressions are shown below:

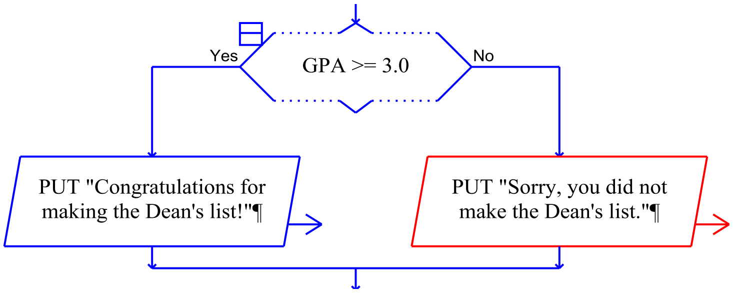
Example	Valid or Invalid?
`(3<4) and (10<20)`	Valid
`(flaps_angle < 30) and (air_speed < 120)`	Valid, assuming `flaps_angle` and `air_speed` both contain numerical data.
`5 and (10<20)`	Invalid - the left side of the "`and`" is a number, not a true/false value.
`5 <= x <= 7`	Invalid - because the `5 <= x` is evaluated into a true/false value and then the evaluation of `true/false <= 7` is an invalid relational comparison.

Selection Control Examples

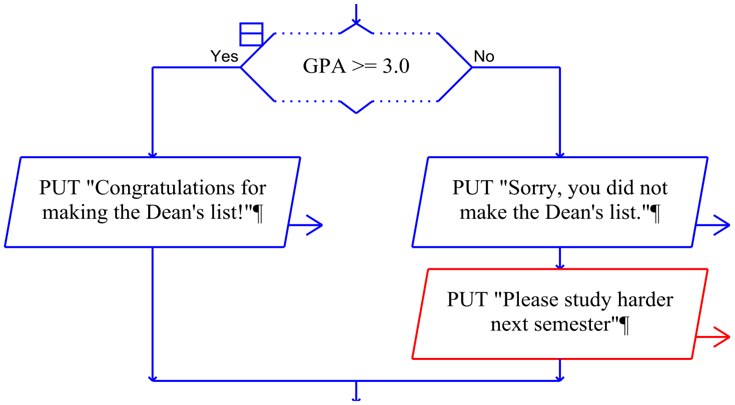
To help clarify selection-control statements, please study the following examples. In the first example to the right, if a student has made the Dean's List, then a congratulations message will be displayed - otherwise nothing is displayed (since the "no" branch is empty).



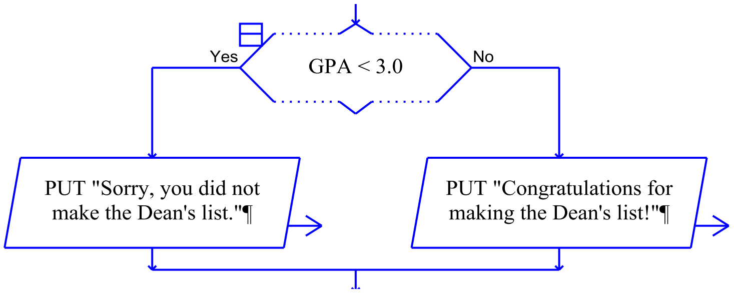
In the next example, one line of text is always displayed, with the value of the PGA variable determining which one.


Yes
GPA >= 3.0
No
PUT "Congratulations for making the Dean's list!"¶
PUT "Sorry, you did not make the Dean's list."¶

In the next example, if the student does not make the Dean's list then two lines of text are displayed, but only one line is displayed if they do.


Yes
GPA >= 3.0
No
PUT "Congratulations for making the Dean's list!"¶
PUT "Sorry, you did not make the Dean's list."¶
PUT "Please study harder next semester"¶

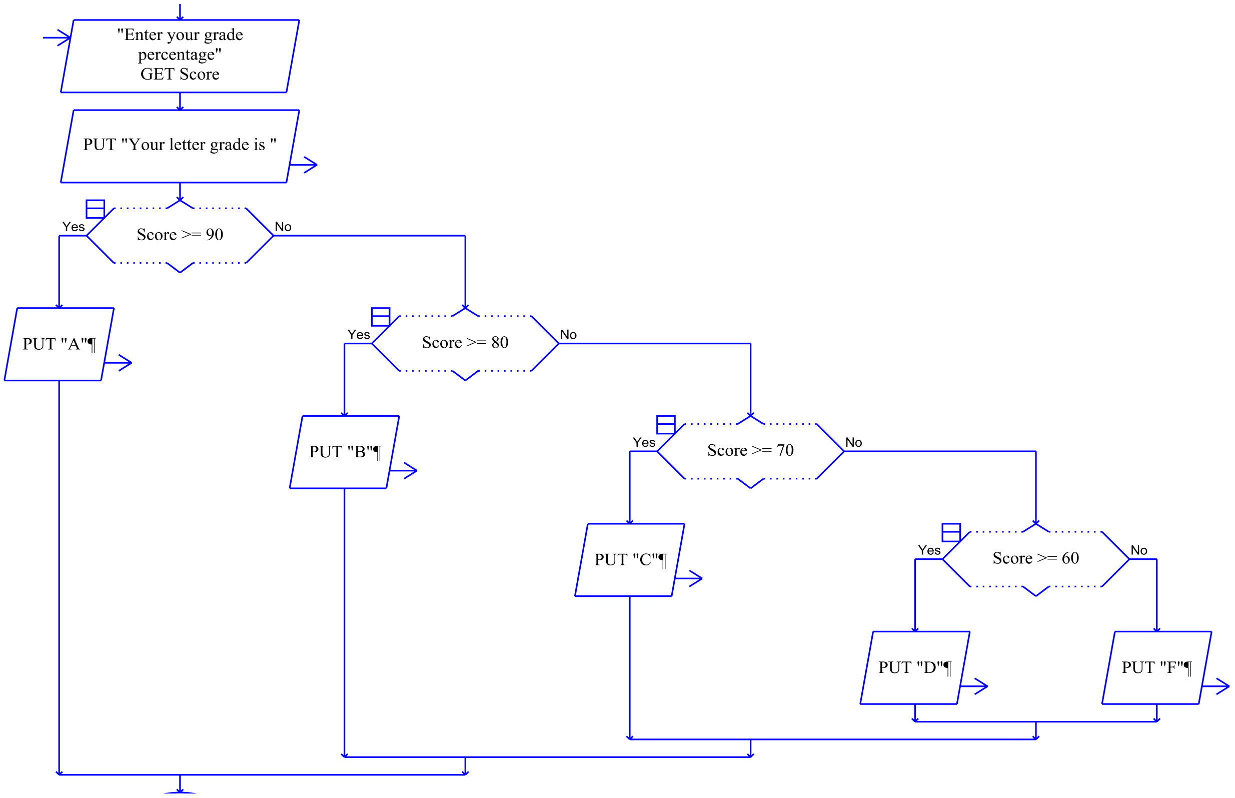
In the last example to the right, the logic of the decision expression has been inverted. This is perfectly acceptable as long as you make sure the inversion covers all possible cases. Note that the inversion of "greater than or equal to" is simply "less than."



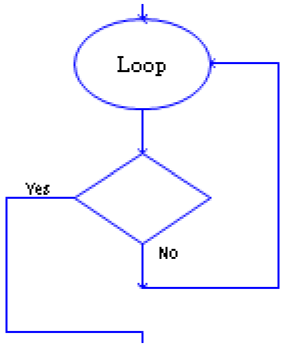
Yes
GPA < 3.0
No
PUT "Sorry, you did not make the Dean's list."¶
PUT "Congratulations for making the Dean's list!"¶

Cascading Selection statements

A single selection-control statement can make a choice between one or two choices. If you need to make a decision that involves more than two choices, you need to have multiple selection control statements. For example, if you are assigning a letter grade (A, B, C, D, or F) based on a numeric score, you need to select between five choices, as shown below. This is sometimes referred to as "cascading selection control," as in water cascading over a series of water falls.



Loop (Iteration) Control

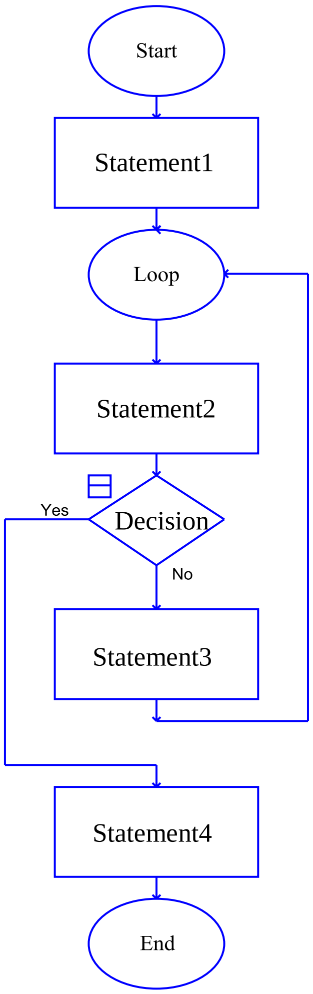


A Loop (iteration) control statement allows you to repeat one or more statements until some condition becomes true. This type of control statement is what makes computers so valuable. A computer can repeatedly execute the same instructions over-and-over again without getting bored with the repetition.

One ellipse and one diamond symbol is used to represent a loop in RAPTOR. The number of times that the loop is executed is controlled by the decision expression that is entered into the diamond symbol. During execution, when the diamond symbol is executed, if the decision expression evaluates to "no," then the "no" branch is taken, which always leads back to the Loop statement and repetition. The statements to be repeated can be placed above or below the decision diamond.

To understand exactly how a loop statement works, study the example RAPTOR program to the right and notice the follow things about this program:

- Statement 1 is executed exactly once before the loop (repetition) begins.
- Statement 2 will always be executed at least once because it comes before the decision statement.
- If the decision expression evaluates to "yes," then the loop terminates and control is passed to Statement 4.
- If the decision expression evaluates to "no," then control passes to Statement 3 and Statement 3 is executed next. Then control is returned back up to the Loop statement which re-starts the loop.
- Note that Statement 2 is guaranteed to execute at least once. Also note that Statement 3 is possibly never executed.



There are too many possible executions of this example program to list them all, but a few of the possibilities are listed in the following table. Make sure you can fill in the fourth column with the correct pattern.

Possibility 1	Possibility 2	Possibility 3	Possibility 4
Statement 1 Statement 2 Decision ("yes") Statement 4	Statement 1 Statement 2 Decision ("no") Statement 3 Statement 2 Decision ("yes") Statement 4	Statement 1 Statement 2 Decision ("no") Statement 3 Statement 2 Decision ("no") Statement 3 Statement 2 Decision ("yes") Statement 4	(do you see the pattern?)

In the RAPTOR example above, "Statement2" could be removed, which means that the first statement in the loop would be the "Decision" statement. Or "Statement2" could be a block of multiple statements. In either case the loop executes in the same way. Similarly, "Statement3" could be deleted or be replaced by multiple statements. In addition, any of the statements above or below the "Decision" statement could be another loop statement! If a loop statement occurs inside a loop, we called these "nested loops."

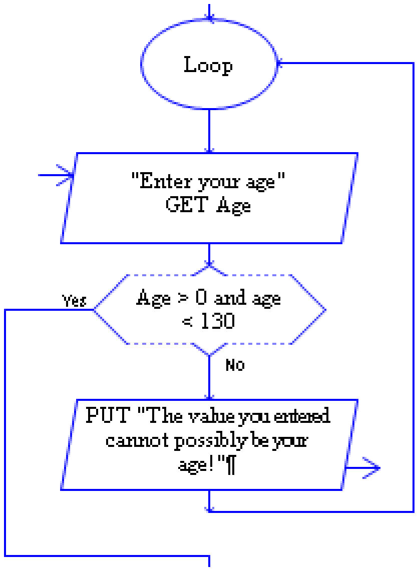
It is possible that the "Decision" statement never evaluates to "yes." In such a case you have an "infinite loop" that will never stop repeating. (If this ever happens, you will have to manually stop your program by selecting the "stop" icon in the tool bar.) You should never write statements that create an infinite loop. Therefore, one (or more) of the statements in the loop must change one or more of the variables in the "Decision" statement such that it will eventually evaluate to "yes."

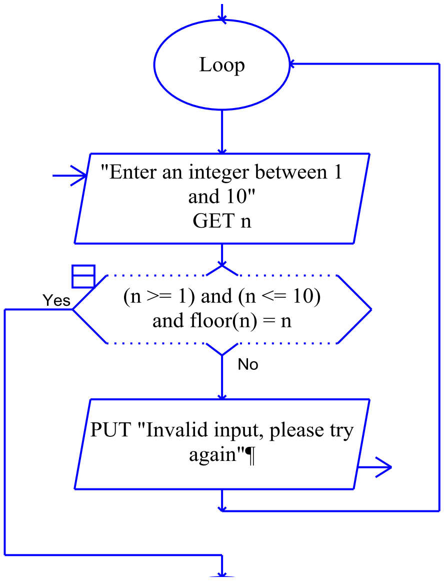
Input Validation Loops

One common use for a loop is to **validate user input**. If you want the user to input data that meets certain constraints, such as entering a person's age, or entering a number between 1 and 10, then validating the user input will ensure such constraints are met before those values are used elsewhere in your program. Programs that validate user input and perform other error checking at run-time are called **robust** programs.

A common mistake made by beginning programmers is to validate user input using a selection statement. This can fail to detect bad input data because the user might enter an invalid input on the second attempt. Therefore, you must use a loop to validate user input.

The two example RAPTOR programs below validate user input. Hopefully you see a pattern. Almost every validation loop that you write will include an input prompt, a decision, and an output error message.




Loop
"Enter an integer between 1 and 10"
GET n
Yes
(n >= 1) and (n <= 10) and floor(n) = n
No
PUT "Invalid input, please try again"¶

Counting Loops

Another common use of a loop is to execute a block of code a specific number of times. This type of loop is called a **counter-controlled loop** because it requires a variable that "counts by one" on each execution of the loop. Therefore, besides the loop statement, a counter-controlled loop requires a "counter" variable that is:

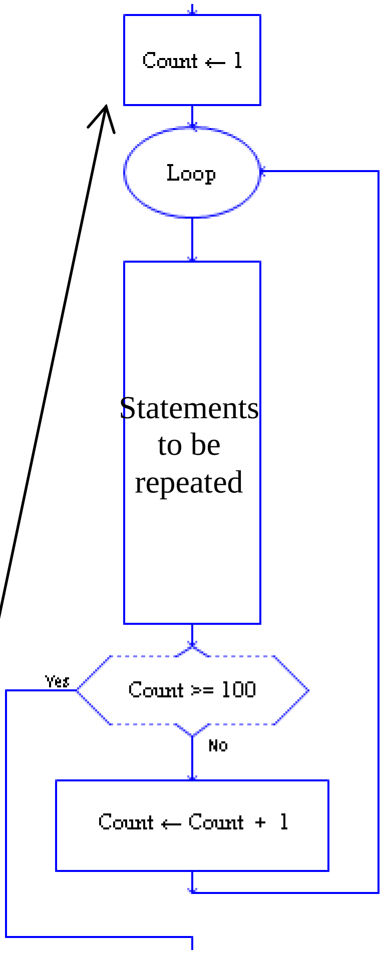
1. initialized before the loop starts,
2. modified inside the loop, and
3. used in the decision expression to stop the loop.

The acronym I.T.E.M (Initialize, Test, Execute, and Modify) can be used to check whether a loop and its counter variable are being used correctly.

An example of a count-controlled loop that executes exactly 100 times is shown to the right. As you study this example, please notice the following important points:

- In this example, the "counter" variable is called "Count." You can use any variable name, but try to make it descriptive and meaningful for your current task.

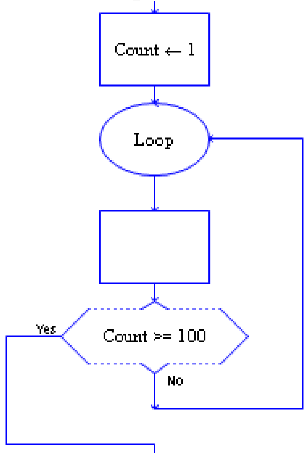
- The "counter" variable must be initialized to its starting value before the loop begins. It is common to start its value at one (1), but you could have a loop that executes 100 times by starting at 20 and counting to 119. Try to use a starting value that is appropriate for the problem you are solving.



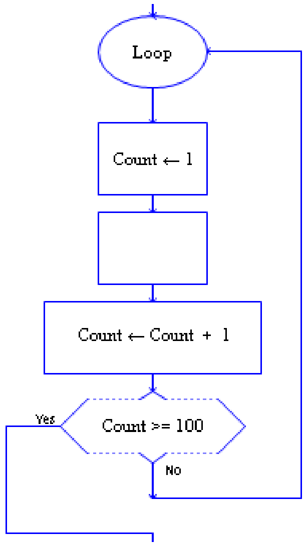
- The decision expression that controls the loop should typically test for "greater than or equal to." This is a safer test than just "equal to."

- A counter-controlled loop typically increments the counter variable by one on each execution of the loop. You can increment by a value other than one, but this will obviously change how many times the loop repeats.

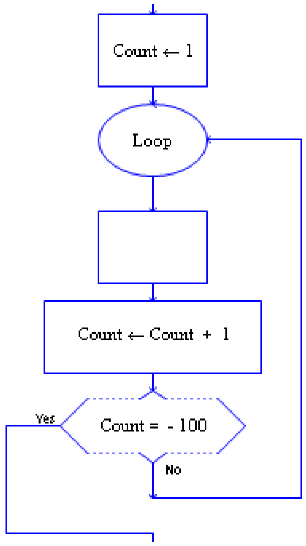
The following three RAPTOR programs demonstrate common errors that should be avoided when implementing loops. See if you can determine the error in each program. (If you can't find the errors, they are explained in footnotes at the bottom of the page.) All three of these problematic programs create an **infinite loop** – that is, a loop that never stops. To avoid writing infinite loops, avoid these common errors.



[1]



[2]



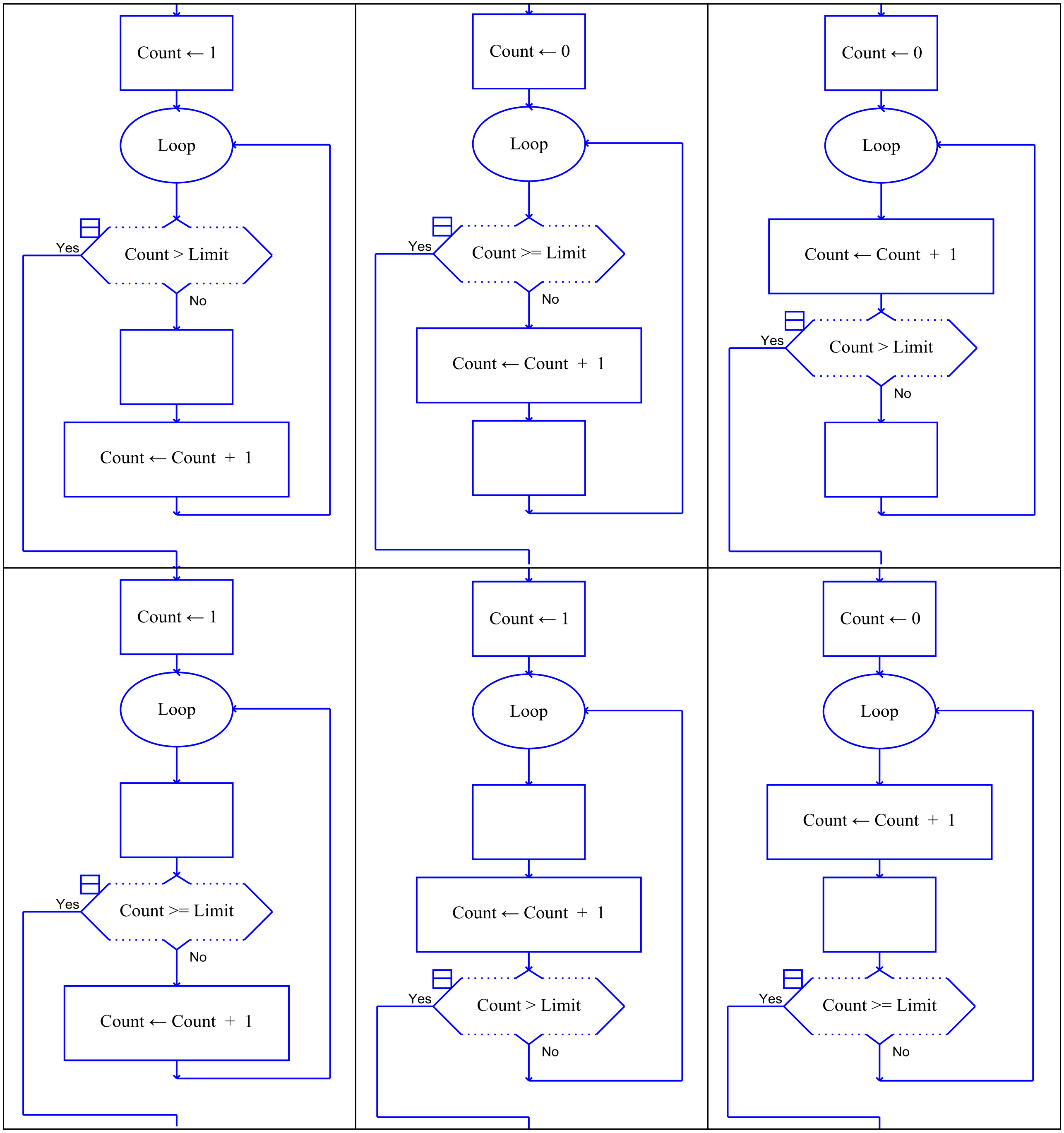
[3]

The following example programs show the six (6) possible variations of a counter-controlled loop. They all do the same thing -- they execute the statement(s) represented by the empty box `Limit` number of times. You can use the variation that makes the most sense to you. In each example, pay close attention to the starting (initial) value of `Count` and the Decision expression.

[1] Count never gets modified and is always equal to 1.

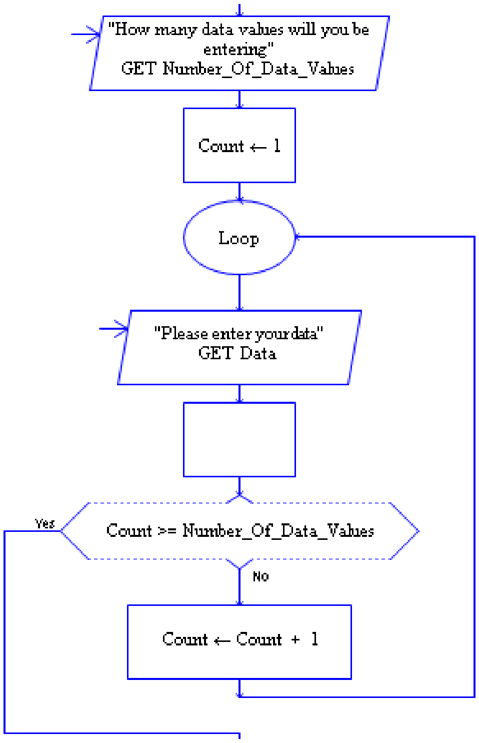
[2] Count is reset to 1 on every execution of the loop and therefore never becomes greater than 100.

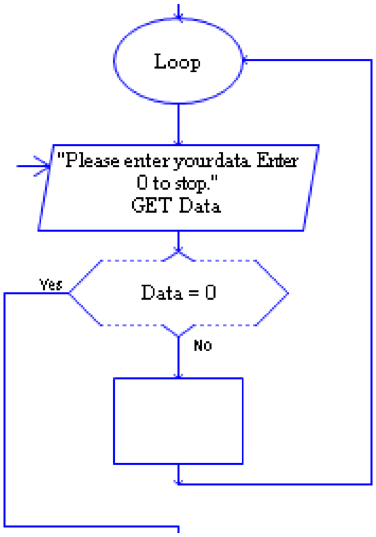
[3] The Decision expression will never evaluate to "yes"


Count ← 1
Loop
Yes
Count > Limit
No
Count ← Count + 1
Count ← 0
Loop
Yes
Count >= Limit
No
Count ← Count + 1
Count ← 0
Loop
Count ← Count + 1
Yes
Count > Limit
No
Count ← 1
Loop
Yes
Count >= Limit
No
Count ← Count + 1
Count ← 1
Loop
Count ← Count + 1
Yes
Count > Limit
No
Count ← 0
Loop
Count ← Count + 1
Yes
Count >= Limit
No

Input Loops

Sometimes you need a user to enter a series of values that you can process. There are two general techniques to accomplish this. The first method is to have the user enter a “special” value that signifies that the user is finished entering data. A second method is to ask the user, in advance, how many values they will be entering. Then that value can be used to implement a counter-controlled loop. These two methods are depicted in the following example programs. In both cases the empty boxes signify where the entered data would be processed. Don’t worry about how the data is processed, just look at these examples to see how the user controls how much data is entered.


"How many data values will you be entering"
GET Number_Of_Data_Values
Count ← 1
Loop
"Please enter your data"
GET Data
Yes
Count >= Number_Of_Data_Values
No
Count ← Count + 1

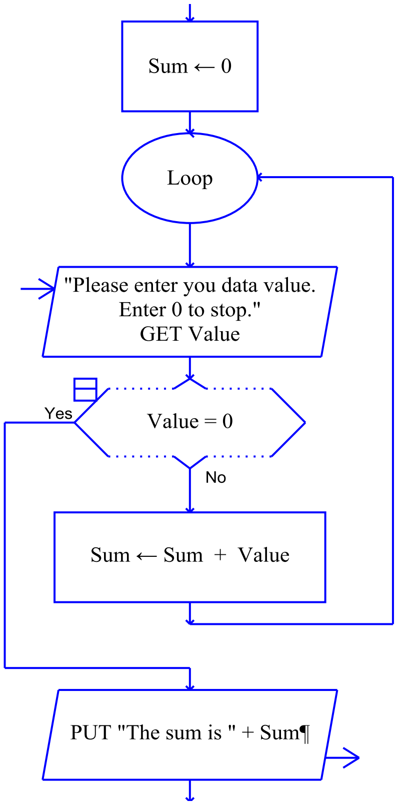


"Running Total" Loops

Another common use of loops is to calculate the sum of a series of data values, which is sometimes called a "**running total**" or a "**running sum**." The example program to the right produces a "running total" of a series of values entered by a user.

To create a "running sum" you must add two additional statements to a loop:

- An initialization statement, before the loop starts, that sets a "running sum" variable to zero (0).
  For example,
  ```
  Sum ← 0
  ```
- An assignment statement, inside the loop, that adds each individual value to the "running sum" variable.



For example,

```
Sum ← Sum + Value
```

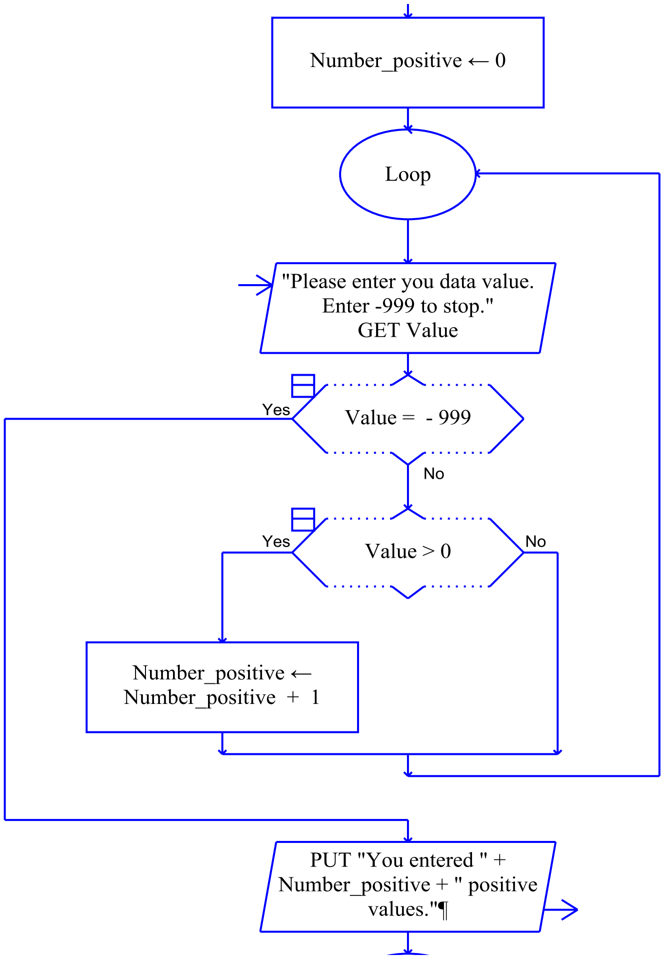
Make sure you understand the assignment statement, `Sum ← Sum + Value`. In English it says, calculate the expression on the right side of the arrow by taking the current value of `Sum` and adding `Value` to it. Then place the result into the variable Sum.

The variable name, `Sum`, is not magic. Any variable name could be used, such as `Total` or `Running_Sum`.

"Counting" Loops

Another common use of loops is for counting the number of times an event occurs. An example of this type of program logic is shown to the right. Note how similar this program is to the previous example.

The last two examples demonstrate how the same pattern of programming statements occurs over and over again and can be used to solve a variety of similar problems. By studying and understanding the simple examples in this reading you will be able to use these examples as the basis for solving additional, more complex problems.



Number_positive ← 0
Loop
"Please enter you data value. Enter -999 to stop." GET Value
Yes
Value = - 999
No
Yes
Value > 0
No
Number_positive ← Number_positive + 1
PUT "You entered " + Number_positive + " positive values."¶

Summary

In this reading we have covered how to write **Selection** and **Loop** statements in RAPTOR. **Selection** statements are used when you need to execute some statements while skipping others. **Loop** statements are used to repeat a block of statements. If you are having a difficult time determining whether to use a **Selection** statement or a **Loop**, it might be helpful to ask yourself the following questions:

Do you need to do something or not do something?	(`Selection`)
Do you need to do one thing or another (but not both)?	(`Selection`)
Do you need to do one of many different things?	(`Cascading Selection`)
Do you need to do the same thing more than once?	(`loop`)
Do you know how many times you must repeat something?	(`count-controlled-loop`)

When developing **Selection** statements it is helpful to keep in mind the following questions:

Will the decision expression cause the correct statements to be executed?
Will the decision expression cause the correct statements to be skipped?

When developing **Loops** in your program it is helpful to keep in mind the following questions:

What statements do I need to repeat?
Have I initialized all variables correctly before the loop starts?
Will the Decision expression always evaluate to "Yes" at some time during execution?
If it is a counter-controlled loop, will it execute the correct number of times? (The most common logic error in programming is an "off-by-one error," which means you want the loop to execute N times, but it executes (N-1) or (N+1) times instead. Always check for off-by-one errors as you are implementing your programs.)

What you have hopefully learned…

- The ordering of programming statements is a key part of program development.

- There are 3 basic types of program flow: Sequential, Selection, and Loop (Iteration).

- Decision expressions, which evaluate to a "Yes"/"No" (true/false) value are used to determine the path a program takes to its "next instruction."

- When to use **Selection** statements and/or **Loop** statements for a particular problem solving task.

- **Selection** statements are used to execute or skip one or more programming statements.

- **Loop** statements are used when one or more programming statements must be repeated.

- The difference between "counter-controlled loops," "input loops," and "running total" loops.

- Infinite loops are bad and special care should be used to make sure your loops always terminate.

Reading Self-Check

Which control structure would be most appropriate for the following problems:
Sequential, Selection, Cascading Selection, or a Loop
_____________ Printing an appropriate message for a cadet's class year.
_____________ Checking for a correct input and continually re-checking if incorrect.
_____________ Computing the average GPA of your CS110 section.
_____________ Determining the volume of a sphere given a radius.
_____________ Initializing all of the variables at the beginning of a program.
_____________ Determining whether a vowel, constant or digit has been typed in.
_____________ Writing out a message if an integer variable contains a negative value.
_____________ Writing "Odd" or "Even" depending on an integer variable's value.
_____________ Writing out the squares of the numbers 1 though 100.
_____________ Reading in scores until a user enters a negative number.

Which of the following Decision expressions will always evaluate to "Yes", always evaluate to "No", or could possibly be either "Yes" or "No"?

_____________ `GR_Score > 100` **or** `GR_Score < 90.`
_____________ `GR_Score > 100` **and** `GR_Score < 90.`
_____________ `GR_Score < 100` **or** `GR_Score > 90.`
_____________ `GR_Score < 100` **and** `GR_Score > 90.`

Write a series of RAPTOR statements that determines if X has the value `1`, `2`, or `3`, and prints out “`ONE`”, “`TWO`”, or “`THREE`” accordingly.

Write a complete program that converts between degrees Fahrenheit and Celsius. The user must first enter the conversion that is desired (F to C or C to F) using any means you want and then enter the value to be converted. The formulas for conversion are:

$F = 9/5\ C + 32$ and $C = 5/9\ (F – 32)$

Write a complete program that plays the game of HI-LO. The program asks one user for a number between 1 and 100 and verifies that such a number has been entered. It then asks a second user for a guess and reads it in. If the guess is correct a congratulation message is written to the screen and the program ends. Otherwise the message “HI” or “LOW” is displayed (if the guess is higher or lower) and another guess is asked for and read in.