Computers & the Humanities 281

Transcript as a Programming Language: Continued

Boolean Operators

It sometimes becomes necessary to compare values within Revolution. Boolean Operators symbolize the relationship between values that can be evaluated to true or false. Transcript recognizes the algebraic symbols: = ,<,> etc., as well as the key words `is` and `is not` as boolean operators.

Comparing Strings: The comparison of words or strings of letters is based on alphabetical order. Consequently, a word that comes before in alphabetical order is considered less than a word that comes after:

- "aaa" is less than "bbb"
- case is ignored: "AAA" is equal to "aaa"
- accents are not ignored: "aaa" is not equal to "äãâ"
- numbers are less than letters: "a1" is less than "aa"

Complex Expressions: The logical operators `and`, `or`, and `not` can be used to build complex logical expressions. Use parentheses to nest or group expressions as necessary. For example:

```
(x = 20) and (y < 45) or (z >= 90)
```

In this example the statement would resolve to true if the first two statements were both true or if the third statement was true. If the third statement was false and one of the first two was false, then the statement would resolve to false.

If Statement

An if-statement is a conditional control structure that allows for checking a certain condition before taking an action. Boolean Operators are an essential ingredient in these types of statements. Transcript has three forms of the if-statement.

Single Statement Form

The most basic form of the if-statement executes a single command when a condition is true. Its syntax is:

```
if <condition> then <statement>
```

where

`<condition>` is an expression that resolves to true or false.
`<statement>` is any single Transcript statement.

Practically speaking, the if-then statement would look something like this within a handler:

```
on mouseUp
   put field "answer 1" into x
   if x = 3 then put "Your answer is correct!" into field "response"
end mouseUp
```

This handler takes the content of a field and puts it into a variable. It then checks to see if the variable's content is equal to 3. If this is true, then an appropriate response is put into another field. If the variable does not contain 3, then the statement is false and the transcript statement after "then" is not executed.

If/Then/Else Form

This expanded form of the if-statement provides an alternate else-clause for a statement to be executed when the condition is false. The syntax is similar:

```
if <condition> then <statement>
else <statement>
```

where

`<condition>` is an expression that resolves to true or false.
`<statement>` is any single Transcript statement.

In a handler, the if/then/else-statement would look something like this:

```
on mouseUp
   put field "answer 1" into x
   if x = 3 then put "Your answer is correct!" into field "response"
   else put "You answer is incorrect!  Try again." into field "response"
end mouseUp
```

Continuing with the example used above, the else portion of this statement provides for everything else the variable may contain besides 3. Consequently, for everything but 3 it will indicate that the response is incorrect.

Multiple Statement Form

The first two forms of the if-statement only allow for one statement in each clause. A third form uses an expanded syntax to overcome this limitation:

```
if <condition> then
   <statement>
   <statement>
   ...
else
   <statement>
   <statement>
   ...
end if
```

where

`<condition>` is an expression that resolves to true or false.
`<statement>` is any single Transcript statement.

In a handler, this form of the if/then/else-statement would look something like this:

```
on mouseUp
   put field "answer 1" into x
   if x = 3 then
      show image "smileyFace"
      put "Your answer is correct!" into field "response"
      wait 2 seconds
      hide image "smileyFace"
      put empty into field "response"
   else
      beep
      put "You answer is incorrect!  Try again." into field "response"
      wait 2 seconds
      put empty into field "response"
   end if
end mouseUp
```

Building upon the example used previously, this handler now does more depending upon the contents of the variable x. If x contains 3 then an image is shown and text is put into a field. The computer waits a few seconds, then hides the image and erases the text. If x contains anything other than 3, then the computer plays the alert sound and gives an apprpriate textual response, erasing the text after a few seconds.

The results of a conditional statement or logical expression can be placed into a variable. This consequently makes the variable contain either true or false (hence, becoming a boolean), enabling the variable to be checked much like a conditional statement. This is desirable in many cases where you would want to confirm a certain condition as being met without evaluating a conditional statement each time. All that needs to be done would then be to check if the variable itself contained either true or false.

To see this in action, imagine that you have some type of exam that contains a listening component. The instructor wishes the students to begin with the listening section before being able to view the pertinent questions. Once they leave the card that gives then access to the sounds, they should not be able to return to listen to it again. This could be accomplished in this manner:

```
on mouseUp
   global heardSound
   if heardSound then
      beep
      put "You've already listened to the sounds." into field "response"
      wait 2 seconds
      put empty into field "response"
      go card "Questions 1-12"
   else
      go card "Listening Comprehension"
   end if
end mouseUp
```

Previously when the stack was opened, the global variable had been given a value with a statement like this: `put false into heardSound` (this variable would then change depending upon the actions of the user). This handler then accesses the global variable and checks its value. Note how this is done (as opposed to `if heardSound = true`). Appropriate action then takes place depending upon the value within the variable. As stated before, it is often more appropriate (and legible) to use a boolean in this manner rather than employing a conditional.

Again, the if-statement is a powerful tool for getting the stack to execute functions on a conditional basis, i.e., when certain conditions are met.

Repeat Statement

The Repeat statement is Transcript's loop structure that allows a series of commands to be repeated under various conditions. When used wisely and judiciously, a repeat loop can be an incredible time-saving control structure as it allows for the quick execution of repetitive tasks.

The general form of the Repeat statement consists of the key word `repeat` at the beginning and a matching `end repeat` at the end, with any number of Transcript statements between.

```
repeat <loop form>
   <statement>
```

```
   ...
end repeat
```

where

<loop form> is an expression that determines the type of loop structure.
<statement> is any single Transcript statement.

There are four basic variations of the loop form:

Repeat For

This specifies a fixed number of times the enclosed statements are to be repeated. A practical example would be:

```
on mouseUp
   repeat for 4 times
      show image "Wow!"
      wait 10
      hide image "Wow!"
      wait 10
   end repeat
end mouseUp
```

Within this handler, this repeat structure would show and hide the image specified (with pauses between each) four times, creating a blinking image. The repeat structure will execute its statements exactly the number of times specified.

Repeat With

This repeats with a counter variable that can be accessed within the loop. The repeat with-statement specifies a variable to use, a start value and an end value. When processing enters the loop the variable is assigned the start value, then is incremented by 1 each time the loop repeats. When it reaches the end value, looping ends. For example:

```
on mouseUp
   repeat with count = 1 to 10
      put count into field "timer"
      wait 1 second
   end repeat
   beep
end mouseUp
```

This handler utilizes the nature of the repeat loop form to change the content of a field. The repeat control structure creates a variable called "count" and initializes it to 1. The first time through the loop it puts the contents of the variable (the value 1) into a field and waits one second. The repeat structure then increments the contents of count by 1. This value (now 2) is then displayed in the field, and so on up until count contains 10, which is the last value displayed in the field. The repeat loop then finishes and the computer plays an alert sound. This particular handler creates a type of ten-second timer that displays the amount of time passed and alerts when ten seconds have passed.

Repeat While and Repeat Until

These are logically reciprocal forms. The English meaning best conveys it: The repeat while-statement repeats the commands inside the loop while a condition *is* (or *remains*) true, and repeat until-statement repeats commands until a condition *becomes* true. For example:

```
on mouseUp
   put 1 into count
   repeat while count <= 10
      put count into field "timer"
      wait 1 second
      add 1 to count
   end repeat
   beep
end mouseUp
```

This handler accomplishes the same end as the example given above. However, in this case it is necessary to manually initialize the variable count outside the loop and manually increment it inside the loop for it to function as we would like.
Note: If "<" were used instead of "<=" within the loop form, then the loop would execute only 9 times instead of 10.

Here is an example of the repeat until-statement:

```
on mouseUp
   put 1 into count
   repeat until count > 10
      put count into field "timer"
      wait 1 second
      add 1 to count
   end repeat
   beep
end mouseUp
```

Again, this accomplishes the same end as the previous examples. It is still necessary to manually initialize and increment the variable.
Note: If ">=" were used instead of ">" within the loop form, then the loop would execute only 9 times instead of 10.

Repeat For Each

This is similar to the Repeat With structure, except that instead of incrementing a variable, this repeat structure allows one to step through the various elements within a container (we will discuss later the exact nature of these elements called "chunks"). This structure is useful if you need to manipulate or perform an action on each element within a container, as it is much faster and easier to work with than the Repeat With structure. For example:

```
on mouseUp
   repeat for each word thisOne in field "fullText"
      put thisOne & return after field "wordList"
   end repeat
end mouseUp
```

This handler allows us to take the text of a given field, step through it one word at a time, and then place each word on a separate line in another field. While the practical uses of this structure may not be immediately evident, its usefulness cannot be underestimated.

Course Schedule
Main Page