**Computers & the Humanities 281**

# Transcript as a Programming Language

Transcript is a full-featured programming language with all the constructs of Pascal, BASIC, C, and other common procedural programming languages. This lecture will explain how Transcript implements these basic constructs. Those who have experience with programming languages will recognize the similarities and differences.

## Variables/Containers

**Variable:** A variable is a named location that can hold a value. It can be visualized as a box in the computer's memory with a name on it. The box can hold some information which can be referenced by the name. A Revolution variable name:

- May be any single string (no spaces) of letters, numbers, or underscores.
- Must begin with a letter or underscore (never a number).
- Cannot be a Transcript key word.
- Is not case sensitive, but capital letters are often useful to improve readability.

To create a variable, follow this structure (the structure is explained in more detail under **Put** in the Command section below):

```
put "3" into holder
```

- **`put`** This command is required.
- **`"3"`** This can be whatever value you wish the variable to contain.
- **`into`** There are other acceptable words.
- **`holder`** This becomes the variable's name by which you may refer to it.

Script it that way and the last item in that command will always be turned into a variable and it will always hold the value of the second item.

Think of creating a variable this way: You've seen and used a trash can. It's a holder for things. Let's create an imaginary trash can and call it "Oscar" (indulge me). If you put a newspaper into the trash can, the newspaper is still a newspaper. You can look inside the can and identify its contents as a newspaper. You can reach inside the can and pull out the newspaper and replace it with an apple core. Oscar now contains an apple core (which retains its identity as an apple core). If you place the newspaper into a different trash can called "Felix," it is still a newspaper. Throughout all this, the trash cans retain their names and you can still refer to them as Oscar or Felix, since the trash cans never lose their names or identities as containers though their contents may change. At the end of our example, then, Oscar is holding an apple core and Felix is holding a newspaper (so apropos).

It's the same way with a variable. If you have a variable and you name it "holder" for example, then you put "3" into that variable, the variable "holder" is just a container and inside it is the number 3. Putting 3 into the variable doesn't change the nature of the 3 and it doesn't change the variable's name.

The first time you state the name of a variable, the box with that name is created. Once the variable has been created, you may use the name to put a value into the variable. You may also use the name to put the value the variable contains somewhere else. The value within a variable disappears whenever Revolution quits.

There are three types of variables in Revolution: local, script local, and global. The difference between these three types exists in how long their value lasts and in their scope. All variables, regardless of their type, lose their contents whenever the application quits.

- **Local:** This type of variable exists only for the handler in which it is created. As soon as the handler finishes executing its commands, the variable disappears. The next time the handler is executed, the variable is created anew and does not retain its value from the previous time the handler was executed. By default, all variables created are local. So in the example above holder would exist as a variable (and consequently its contents) only for the duration of the handler in which it was created.
- **Script Local:** This type will not be covered in this course. You may refer to the Revolution documentation for further information.
- **Global:** This type of variable is accessible from any handler anywhere in the stack. Unlike a local variable, a global variable retains its value even after the execution of the handler that created it. A global variable must be declared as such:

  ```
  global someGlobalVar
  ```

  A global variable must be declared and identified as such in every handler that uses that global variable, and the declaration must occur in the handler before the line where the global variable is used. Otherwise, the handler will instead create a local variable with the same name. For example:

  ```
  on mouseUp
     global holder
     put 12 into holder
  end mouseUp
  ```

  When this handler is executed, holder is declared to be global variable and is given the value of 12. This value would then be accessible to all handlers which access holder as a global variable.

  It is also possible to declare a global variable outside all handlers within a particular script. This would make the variable accessible to all handlers within that script and would obviate the need to declare it within each handler.

**Container:** A container is anything that possesses a name and holds a certain value or string. By this definition, then, a variable is also known as a container. A Revolution field has a name and can hold values and strings, and is therefore a container. In this respect a field is similar to a variable but differs in that it has a place on the card, not just in memory. Consequently, values stored in fields become part of the stack and do not disappear once the

application quits (if you've saved the stack). The term "container" in Revolution therefore encompasses variables and fields to refer to both. Another way to think of this is that variables and fields are subsets of the the larger group of containers.

**It:** This is a special-purpose Transcript variable/container to hold the results of the `get` command (and several other commands you'll learn about later). It also serves to make scripts more readable. However, you need to be careful as the `it` variable is not the same as the It pronoun.

**Message Box:** The message box itself is a container. Consequently, you can do the same things with the message box as with other containers:

- Put something into it.
- Put the contents of it somewhere else.
- Put the result of an expression into it.

As explained earlier, if no destination is specified for a `put` command it goes into the message box by default. The contents of the message box are accessed by the special Revolution word `msg`.

## Constants/Literals

**Constant:** Like variables, constants have a name and contain a value, but unlike variables, that value does not change. Constants are just that: constant and fixed. They are neither changeable, nor fluid, nor mutable, nor protean. The usual purpose of constants is to provide useful values that are difficult to do as literals. Here are some of the more useful literals:

- **one - nine:** These don't provide much advantage over the digits 1-9 except perhaps readability.
- **pi:** This gives the value of pi to 20 decimals.

We will encounter more later. Be very scared.

**Literal:** A literal is something that is the value itself. In Transcript, anything enclosed in double quotes is treated as a literal. This becomes extremely important in keeping literals distinguished from variables. A variable with nothing in it often (but not always) functions like a literal. This can cause considerable confusion. Look at this handler, for example:

```
on mouseUp
   put "flower" into thisImage
   show image thisImage
end mouseUp
```

This would take the literal string "flower" and place it within a variable called "thisImage." In referencing the variable on the very next line, the handler actually looks at the contents of the variable. Therefore, the handler would display an image called "flower." However, the handler would operate differently if one line were changed minutely:

```
on mouseUp
   put "flower" into thisImage
   show image "thisImage"
end mouseUp
```

Here the second command of the handler would display an image called "thisImage" since enclosing thisImage within quotation marks indicates that it is to be treated as a literal and not as a variable. It is therefore good practice to always use quotes around the names of all objects when referring to them by their names to avoid problems.

## Operators

**Operator:** An operator specifies an operation to be performed on its operands. You are probably most familiar with arithmetic operators:

- **+** and **-** for addition and subtraction
- ***** and **/** for multiplication and division
- **^** for exponentiation

Transcript follows the standard order of precedence in aritmetic operators you learned in algebra:

1. exponentiation
2. multiplication and division
3. addition and subtraction

As in algebra, parentheses can be used to override default precedence.

Some operators are keywords rather than symbols:

- **div** This is the integer divide operator. It divides the first number by the second without remainder. No rounding takes place. The remainder is simply discarded.

  ```
  put 25 div 8 into myNumber
  ```

  This divides 25 by 8 and puts the result of 3 into the variable myNumber. The remainder of 1 is ignored and lost in the digital ether.

- **mod** This is the modulo operator. It divides the first number by the second and returns the remainder. The rest is discarded.

  ```
  put 25 mod 8 into myNumber
  ```

This divides 25 by 8 and puts the remainder of 1 into the variable myNumber. The rest of the equation is ignored and lost in the digital ether.

We will see other operators later. Be alert. The world needs more lerts.

## Commands

**Put:** The put command is used to put values into containers. You accomplished this when you put various strings into a field. As shown above it works essentially the same for variables. In this regard it is designated as Transcript's assignment statement. The following key words determine how the new value affects the existing contents of the container:

- **into:** This replaces what was there inside the container.

```
on mouseUp
   put "Eeyore" into hundredAcreWood
   put "Pooh Bear" into hundredAcreWood
end mouseUp
```

  This handler will take the literal string "Eeyore" and place it into the variable hundredAcreWood. Then it will place the another literal string "Pooh Bear" into the same variable, replacing the previous contents completely ("Typical," as Eeyore would grumble).
- **after:** This places the new value directly after the current contents.

```
on mouseUp
   put "Eeyore" into hundredAcreWood
   put "Pooh Bear" after hundredAcreWood
end mouseUp
```

  This handler will place the literal string "Pooh Bear" directly after the string already residing within the variable, resulting in "EeyorePooh Bear" contained within hundredAcreWood (when genetic cloning turns awry).
- **before:** This places the new value directly before the current contents.

```
on mouseUp
   put "Eeyore" into hundredAcreWood
   put "Pooh Bear" before hundredAcreWood
end mouseUp
```

  This handler will place the literal string "Pooh Bear" directly before the string already residing within the variable, resulting in "Pooh BearEeyore" contained within hundredAcreWood (more threatening than a heffalump or woozle).

**Arithmetic Commands:** Transcript provides four commands for common arithmetic calculations:

- add
- subtract
- multiply
- divide

These commands perform the calculation on the value within the container and replace the value with the answer. Most programming languages do not have arithmetic commands like these, but accomplish the same result with an assignment statement that references the variable in the calculation and assigns the result to the same variable. This technique also works in Transcript.

## Functions

A function performs a prescribed process and returns a value. Functions are used within the context of a command and may be used anywhere the value it returns is appropriate.

**Forms:** Revolution functions generally have two forms:

- Algebraic form: `sqrt(25)`
- Long/readable/"prose" form: `the sqrt of 25`

**Argument:** An argument is an input value for the function to work on. Most (but not all) functions require one or more arguments. With the example given above (`sqrt(25)`), the argument would be `25`. Arguments may be specified explicitly (as above) but can also come from a variable or a field or other source of value. For example:

```
on mouseUp
   global myNum
   put abs(myNum) into field "Result"
end mouseUp
```

In this example, the argument passed to the function (in this case the absolute value) is a variable. The function will perform its operations on the contents of the variable and place the result in a certain field. Obviously, the arguments passed to a function can vary and the values the function returns are dependent upon the arguments passed.

**Random:** This is a function that can often be used in instructional applications to provide variety and unpredictability. It returns a random value between 1 and the argument, inclusive. For example:

```
on mouseUp
   put random(20)
end mouseUp
```

This handler would then put into the message box any number between 1 and 20, the lowest possible value being 1, and the highest possible value being 20.

We will see many more functions later. You are welcome to explore and experiment with some of them. They are always identified as functions in the Transcript Language Dictionary.

## Concatenation

This is the operation of appending text together, "adding" (disregarding the arithmetic implications of the word) one chunk of text to another. The ampersand (**&**) is the concatenation operator/symbol in Transcript.

- **&** This concatenates directly, i.e., no space between the chunks of text.

  ```
  put "This" & "is" & "my" & "stack" into myStack
  ```

  This results in a string "Thisismystack" being placed in the variable myStack.

- **&&** This concatenates with a space between the chunks of text.

  ```
  put "This" && "is" && "my" && "stack" into myStack
  ```

  This results in a string "This is my stack" being placed in the variable myStack.

The comma (**,**) is a special concatenation symbol in Transcript. For example:

```
on mouseUp
   put "Eeyore","Pooh Bear" into hundredAcreWood
end mouseUp
```

This handler would place the two strings together with a comma separating them (resulting in `Eeyore,Pooh Bear`) into a variable. This symbol is an effective tool for concatenating several strings into a single comma-delimited string.

**Text Constants:** Transcript defines several text constants to provide special characters that cannot readily be typed in directly.

- **quote** This gives a quotation (") mark. Otherwise any quotation marks you include in a command are interpreted as literal delimiters. Note that curled quotes are no problem for Revolution and can be typed in directly.
- **return** This provides a carriage return (referring to the operation of a typewriter of going to the next line on the paper), a character which you cannot create. Hence, this constant.
- **space** This creates a space within a string. It can also be typed, but makes the handler more explicit.
- **tab** This provides a tab, although this version of Revolution doesn't do anything with it. It just looks like a space. However, if you export the text, the tab characters are recognized as such.
- **empty** This gives nothing. It is used primarily to empty out a container.

Course Schedule
Main Page