Computers & the Humanities 281

Messages in Revolution

All Transcript programming is based on the concept of messages. When things happen in the Revolution environment, messages are generated and passed through the hierarchy much like a compulsive play-by-play announcer describing everything that happens in the sporting arena. As a programmer, you can write handlers to intercept such messages and subsequently perform different commands and functions. The syntax for utilizing messages sent in Revolution is similar for all:

```
on <message>
   <statement(s)>
end <message>
```

where

`<message>` is any message generated in Revolution.
`<statement(s)>` is one or more Transcript statements or commands designed to be associated with the message and subsequently executed.

Mouse Messages

There are several messages associated with mouse clicks:

- **mouseUp:** You have already seen the mouseUp message. This is the most commonly used message handler in object scripts. The mouseUp message is sent on the *up* part of a click.
- **mouseDown:** The mouseDown message is sent on the *down* part of a click. Thus every click generates first a mouseDown message, then a mouseUp. The mouseDown and mouseUp handlers are often used in pairs to do one thing when the mouse button is clicked down and something else when it is released.
- **mouseRelease:** The mouseRelease message is sent to the control that was clicked when the user releases the mouse *outside* the clickable area of the object.
- **mouseDoubleUp:** The interval between clicks that determines what qualifies as a double click (as opposed to two single clicks) is set by the operating system. If the second click is within this time interval, the mouseDoubleUp message is sent on the *up* part of the second click.
- **mouseDoubleDown:** The interval between clicks that determines what qualifies as a double click (as opposed to two single clicks) is set by the operating system. If the second click is within this time interval, the mouseDoubleDown message is sent on the *down* part of the second click. Every double click generates first a mouseDoubleDown message, then a mouseDoubleUp.
- **mouseStillDown:** The mouseStillDown message is sent continuously when the mouse button is held down after the initial mouseDown. A mouseStillDown may be used to do something repetatively as long as the mouse is held down.

There are four other messages associated with the mouse. These key on just the cursor and require no click:

- **mouseEnter:** This message is sent when the cursor enters the clickable area of an object (which usually, but not always corresponds to the rectangular area of the object).
- **mouseLeave:** This message is sent when the cursor leaves the clickable area of an object. The mouseEnter and mouseLeave messages are also often paired together as they represent reciprocal actions.
- **mouseWithin:** The mouseWithin message is sent repeatedly after the initial mouseEnter while the cursor remains within the rectangular area of an object. A mouseWithin handler can be used much like mouseStillDown.

  The mouseWithin message alternates with the mouseStillDown message if the mouse button is being held down.

- **mouseMove:** The mouseMove message is sent whenever the user moves the mouse.

Theoretically, these messages can be handled by any object with a proper handler. While this is generally true, reality sometimes paints a different picture. Some of the objects may not handle messages exactly the same as the other objects. You'll have to experiment to discover what works and what doesn't.

Open/Close Messages

Open and close messages are sent when objects are opened and closed (so to speak). There are three objects in Revolution that can utilize these generated messages:

- **openField** & **closeField:** The openField message is sent to an unlocked field when you click or select text within that field. The closeField message is sent to a field when the cursor is removed from that field *and* the field's contents have been changed. If there has been no change made to whatever the field contains, then no closeField message is sent. In this case an **exitField** message is generated and sent to the field.
  Note: The open/close messages are cursor-related and occur when the field's lockText property is false (Lock Text off). *Locked* fields receive mouse messages and can consequently have all the mouse message handlers that buttons can have and will behave accordingly.
- **openCard** & **closeCard:** The openCard message is sent to a card when you arrive at that card, and the closeCard message is sent to the card when you leave for another card.
- **openStack** & **closeStack:** The openStack message is sent to the destination card right after you open a stack. The closeStack message is sent to the current card when the stack closes (either by quitting the application or by closing the stack window).

Open handlers are typically used to do initial setup, music, animation, etc., for when the object first appears. Close handlers are typically used to save answers and generally clean up the environment after the current user.

There are a number of supplementary messages related to the opening and closing of objects:

- **resumeStack** & **suspendStack:** These two messages occur when you have more than one stack open at the same time and switch from one to the other. They are similar in function to openStack and closeStack and are therefore used in conjunction with each other. The resumeStack message is sent to the current card when a stack window is brought to the front. The suspendStack message is sent to the current card when something makes its stack no longer the active window.
- **preOpenCard** & **preOpenStack:** These messages behave just like the open messages described above with one important distinction: These can be used to update the environment before the object actually appears on the screen. These are then useful for performing functions which you do not wish the user to observe.

Keyboard Messages

This class of messages is sent to the card (or to the active/focused control) when keys on the keyboard are pressed:

- **returnKey:** This message is sent to the active object where there is no text selection (i.e., an unlocked field where text editing supersedes) and the user presses the Return key.
- **tabKey:** This message is sent when the user presses the Tab key.
- **enterKey:** This message is generated when the Enter key is pressed, except when typing in an unlocked field in which case you get closeField.
- **arrowKey:** This message is sent when one of the arrow keys is pressed. The message passes a parameter that indicates which key was pressed.
- **functionKey:** This message is sent when one of the function keys is pressed, passing a parameter that indicates the number of the key pressed.
- **keyDown:** This message is sent when other keys on the keyboard are pressed. The parameter passed indicates which key. If this handler exists anywhere in the message hierarchy for an unlocked field, then no typing takes place within the field, unless the message is passed.

Pass

For most Transcript messages we've talked about, when there is no handler to handle the message as it passes through the hierarchy, it gets ignored. However, there are a few Transcript messages that have a default behavior:

- `tabKey` moves the insertion point into first (next) editable field.
- `arrowKeys` navigates the user forward/backward between cards.

When there *is* a handler for a message like arrowKey, it intercepts the message and overrides the default behavior. After Revolution has executed a handler, it considers its task finished and the message which triggered the handler consequently dies. In some cases this may be what you want. Writing an arrowKey handler will prevent users from using arrow keys to navigate into parts of your stack you want to keep secure. However, sometimes you actually want Revolution to execute a object's handler and then perform its default behavior or execute another handler located further up in the hierarchy along the message path right after that. To make that happen, you would do something similar to this in the object's script:

```
on mouseUp
   beep
   pass mouseUp
end mouseUp
```

The pass command tells Revolution to handle the message, then look to the next level of the hierarchy along the message path for a mouseUp handler (be that in a group or a card script). If an appropriate handler is found, it will then execute that handler in the group or card script and stop moving up the levels unless there is also a pass command in that handler.
Note: When the pass command is executed, any remaining statements in the handler are skipped, so this command is best employed at the end of a handler or within an if-control structure.

Send

This command is used to send a message to an object. In this respect it is similar to the pass command. However, with send you can override the normal message path, allowing greater flexibility with where messages are sent. It also differs from pass in that once the command is executed and the message is sent, the rest of the handler is still executed. The basic syntax is:

```
send <message> to <object>
```

where
`<message>` is any message.
`<object>` is any object.

A practical example of this command would look something like this:

```
on mouseUp
   hide image "flowers"
   send mouseUp to button "Reset"
   show field "Instructions"
end mouseUp
```

This handler would first hide a particular object, then send the mouseUp message (it could any given message) to a particular button (or any other given obejct). That button would then execute the mouseUp handler in its script. After that had finished, the original handler would continue executing its handler and would show a particular object. The strength of this command lies in its ability to provide access to handlers in objects outside the normal message hierarchy.

Messages Exercise
Course Schedule
Main Page