Automator AppleScript Actions Tutorial

Apple Applications > Automator

2007-07-18

Contents

Introduction **Introduction to Automator AppleScript Actions Tutorial** 7

Organization of This Document 7

Chapter 1 **Before You Start** 9

What Is an Automator Action? 9
The Action You Will Create 11

Chapter 2 **Creating the Project** 13

Chapter 3 **Creating the User Interface** 19

Opening the Action Nib File 19
Placing and Configuring User-Interface Objects 20

Chapter 4 **Establishing Bindings** 27

Bindings in an Action 27
Establishing the Bindings of the Action 28
Alternatives to Bindings 31

Chapter 5 **Configuring the Action** 33

Editing the Information Property List 33
Action Input and Output 35
Default Parameter Settings 37
The Action Description 38
Other Settings 39

Chapter 6 **Writing the Action Script** 41

The on run Command Handler 41
Writing the Subroutines 43

Chapter 7 Building and Testing the Action 45

Document Revision History 49

Figures

Chapter 1 Before You Start 9

Figure 1-1 The Find Messages in Mail action 9
Figure 1-2 The Copy Unread Mail to iPod Notes workflow 10
Figure 1-3 The Pass Random Items action in a workflow 11

Chapter 2 Creating the Project 13

Figure 2-1 Selecting the AppleScript action project template 14
Figure 2-2 Specifying project name and location 15
Figure 2-3 The files of an AppleScript action project in Xcode 16

Chapter 3 Creating the User Interface 19

Figure 3-1 The nib file window, initial action view, and palette 19
Figure 3-2 Automator icon 20
Figure 3-3 Placing a text field by dragging it from the palette 21
Figure 3-4 Setting the size attribute of the text field 22
Figure 3-5 Resizing the text field 23
Figure 3-6 Changing the string in a in a non-editable text field (a label) 23
Figure 3-7 Changing the number of radio buttons in a matrix (radio buttons) 24
Figure 3-8 Changing the title of a button 24
Figure 3-9 Final user interface of Pass Random Items action 25

Chapter 4 Establishing Bindings 27

Figure 4-1 Binding between the pop-up list and a property of the parameter dictionary 28
Figure 4-2 Adding keys as attributes of the Parameters instance. 29
Figure 4-3 Binding between the controller and the parameters dictionary 30
Figure 4-4 The selectedIndex attribute in the bindings inspector 31

Chapter 5 Configuring the Action 33

Figure 5-1 The property inspector for Automator actions (general collection) 34
Figure 5-2 The completed General collection of Automator properties) 35
Figure 5-3 Automator property inspector—Input collection 36
Figure 5-4 The default parameters for the Pass Random Items action 38

Figure 5-5 The description for the Pass Random Items action 38

Chapter 6 Writing the Action Script 41

Figure 6-1 The template for the `on run` handler 41
Figure 6-2 Initializing local output and parameter variables 42
Figure 6-3 The final `on run` handler 42
Figure 6-4 Subroutines called by the main script 43

Chapter 7 Building and Testing the Action 45

Figure 7-1 Executable settings for the Pass Random Items action 45
Figure 7-2 Testing the Pass Random Items action in a workflow 46
Figure 7-3 The AppleScript debugger 47

Introduction to Automator AppleScript Actions Tutorial

This tutorial shows you how to create an action for the Automator application using AppleScript as the implementation language.

Important: Some of the Xcode features mentioned in this tutorial, such as the AppleScript debugger and the property inspector, were introduced with Xcode 2.1. If you have an earlier version of Xcode, you do not have access to these features.

The tutorial assumes that you are familiar with AppleScript, but otherwise has no other prerequisites. It is helpful, however, if you have some experience with AppleScript Studio.

Organization of This Document

This tutorial has the following chapters, which are meant to be read in the given order:

1. "Before You Start" (page 9) gives an overview of Automator actions and workflows. It also describes the action that you will create in the tutorial.
2. "Creating the Project" (page 13) explains how to create an AppleScript action project and identifies the key elements of such projects.
3. "Creating the User Interface" (page 19) shows you how to create the user interface of the action using the Interface Builder application.
4. "Establishing Bindings" (page 27) explains what Cocoa bindings are and describes how you establish bindings for the action.
5. "Configuring the Action" (page 33) discusses how to set the properties of the action in its `Info.plist` file.
6. "Writing the Action Script" (page 41) shows the script that you write for the action and explains the general structure and behavior of all such scripts.
7. "Building and Testing the Action" (page 45) describes techniques for testing and debugging an action after it is built.

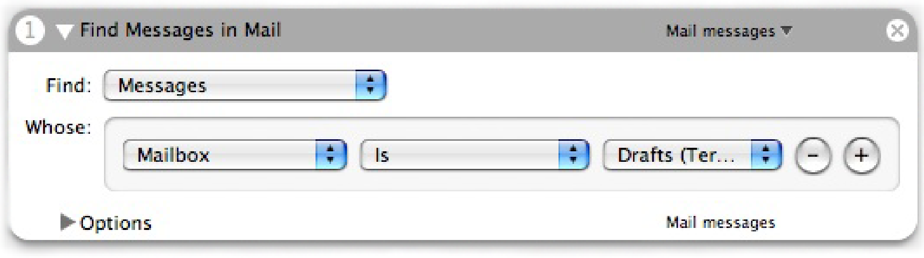
Before You Start

In this tutorial you are going to learn the basic steps for constructing an Automator action using AppleScript as the development language. In the process of learning, you will create a action that you can productively use in workflows. But before you begin, let’s take a moment to review what an action is and to look at the action you will create.

What Is an Automator Action?

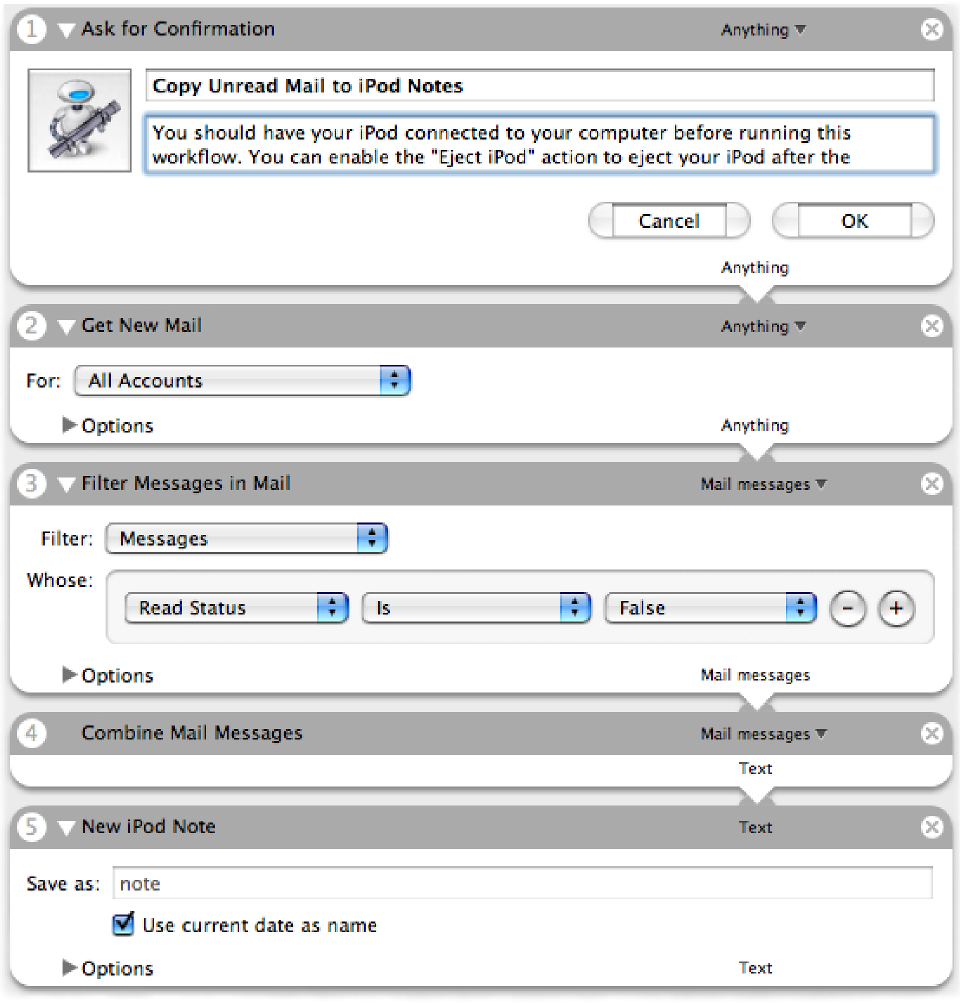
Most people are familiar with the notion of building blocks. By placing small but well-defined units in certain relationships with each other, one can compose complex and even elegant structures. An Automator action is such a building block. An action is a small, discrete functional unit; it performs a well-defined operation usually on data of a specific type, such as copying a file or adding photos to an iPhoto album. It often offers the user a simple user interface for setting certain parameters of the operation. For example, the action in Figure 1-1 selects certain Mail messages based on specified criteria.

Figure 1-1 The Find Messages in Mail action



By itself, an action cannot do much. For one thing, it requires the Automator application to provide the context for its execution. But, more importantly, an action’s very discreteness limits its usefulness; an action is designed to complete a small, well-defined task, and nothing more. To be effective, an action must be placed in a meaningful sequence of other actions. This sequence of actions is called a workflow. In a workflow the output of one action is usually (but not always) passed to the next action in the workflow as input. Automator orchestrates this process by starting each action in turn and passing it the output of the previous action. A workflow expresses a operation that can be arbitrarily complex, and the final product of that operation is usually the output of the last action. For example, the workflow in Figure 1-2 gets a user's unread mail and downloads the messages to the Notes section of a connected iPod.

Figure 1-2 The Copy Unread Mail to iPod Notes workflow



As an Automator workflow (such as the one above) illustrates, an action is usually designed to accept input and produce output of specific data types (although some actions will take and provide any type of data). Thus some actions may be incompatible with other actions; the Combine Mail Messages action, for instance, could not accept Address Book data. But there can be what are called conversion actions to bridge between actions with incompatible types of data.

From a development perspective, an action is a loadable bundle installed in one of four locations:

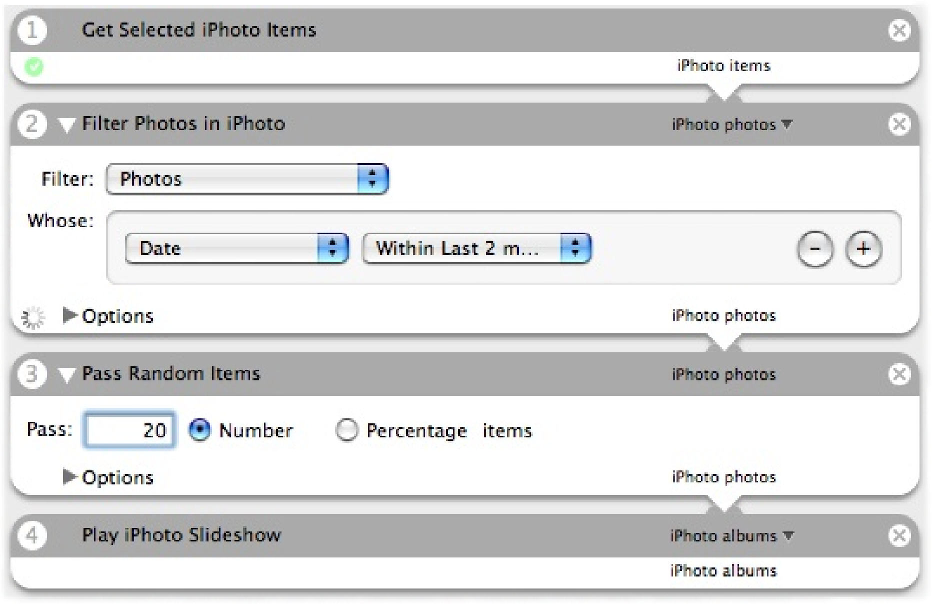
- `/System/Library/Automator` (Apple-provided actions)
- `/Library/Automator` (third-party actions, general access)
- `~/Library/Automator` (per-user access)
- Inside an application bundle (access determined by access to application)

The action bundle can contain executable code, AppleScript scripts, shell scripts, and localized strings, nib files, and other resources. When Automator is launched, the application extracts configuration information from the action bundles and displays some of this information in its user interface. It also loads the bundles of the actions placed in a workflow at some point before the execution of the workflow (the exact moment differs for actions based on Objective-C and actions based on AppleScript). For a complete description of the architecture of Automator actions and workflows, see "How Automator Works" in the *Automator Programming Guide*.

The Action You Will Create

In this tutorial you will create an action named Pass Random Items. The action accepts a list of items (of any type) from the previous action and passes a random subset of those items to the next action. Users can specify the number or percentage of items to pass in the action's user interface. Figure 1-3 shows the Pass Random Items action in a workflow.

Figure 1-3 The Pass Random Items action in a workflow



Note: The project for the Pass Random Items action is installed as an example (under the name Randomizer) in `/Developer/Examples/Automator`.

In this workflow, the Filter Photos in iPhoto action passes all selected photos taken within the last two months to Pass Random Items. This action, in turn, passes 20 random photos from that initial selection to an action that plays them in a slide show.

After you complete this tutorial and before you attempt developing any action on your own, you should take time to consider the design of the action. Read "Design Guidelines for Actions" in *Automator Programming Guide* for a summary of pertinent design guidelines.

Creating the Project

When you create an AppleScript action project, you start by selecting an Xcode template that provides all necessary project files and initial project settings.

The steps for creating a AppleScript action project are few and simple:

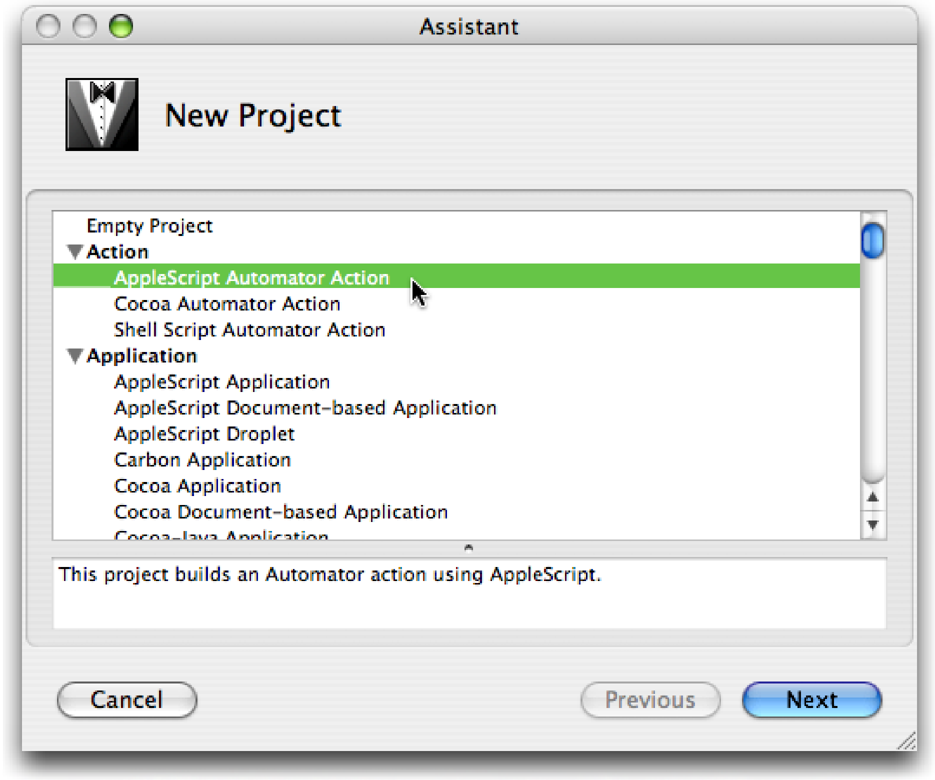
1. Launch the Xcode application.

   You can find Xcode in `/Developer/Applications`.

2. Choose New Project from the File menu.

Xcode displays the New Project assistant (see Figure 2-1). The Automator action project templates are near the top of the displayed list.

Figure 2-1 Selecting the AppleScript action project template



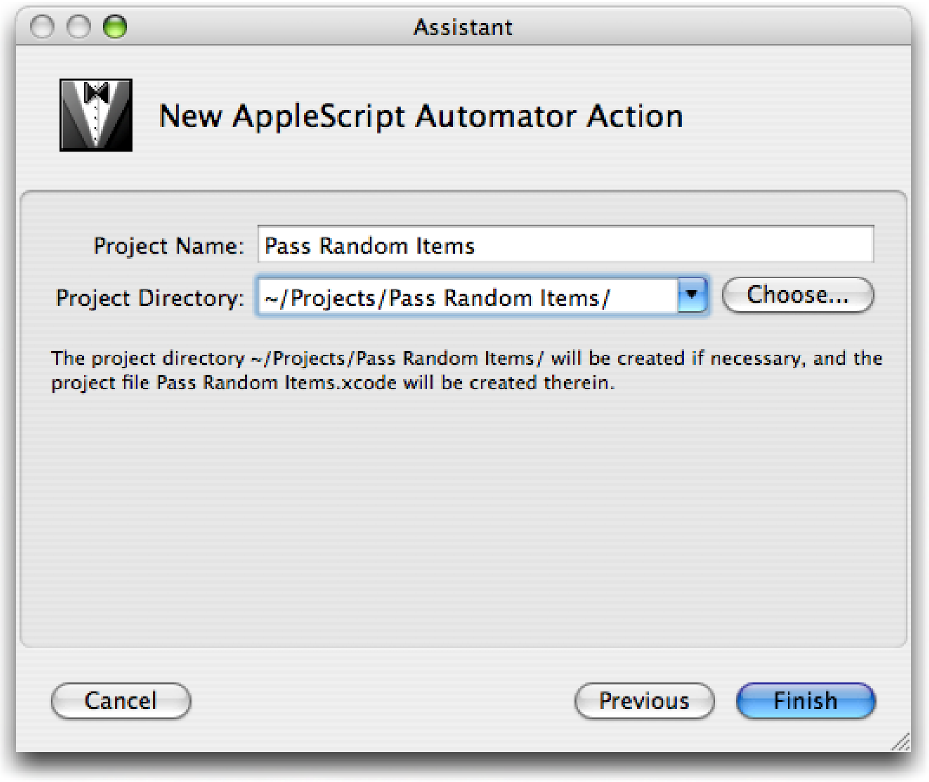
This project builds an Automator action using AppleScript.

3. Select AppleScript Automator Action and click Next.

4. In the New AppleScript Automator Action assistant, enter a project name and select a file-system location for the project (see Figure 2-2).

For the tutorial project, the project name is Pass Random Items.

Figure 2-2 Specifying project name and location



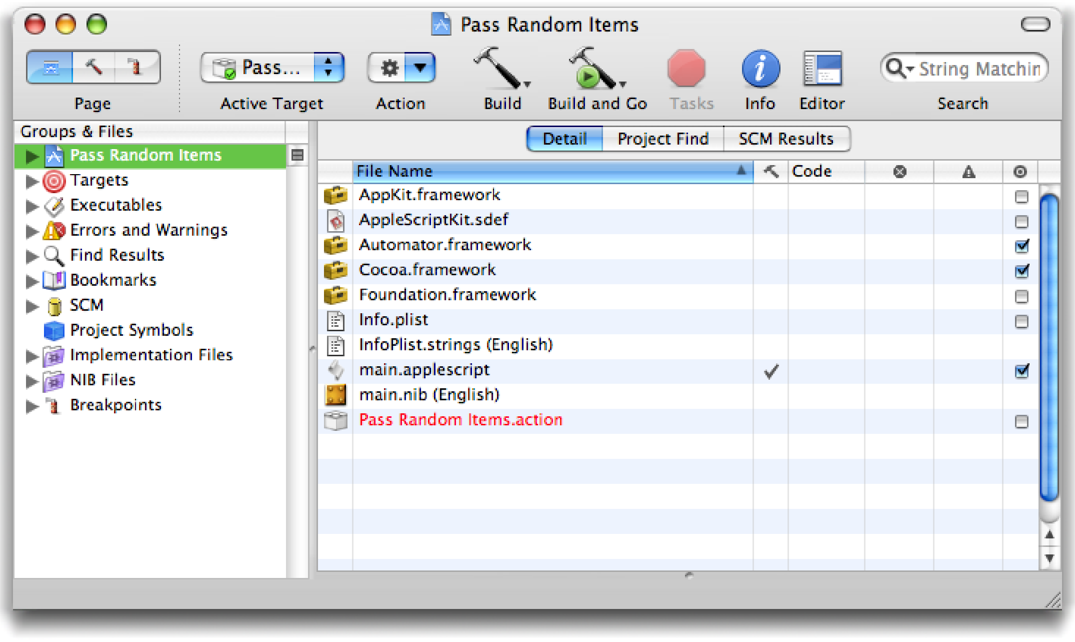
Project Name: Pass Random Items

Project Directory: ~/Projects/Pass Random Items/

The project directory ~/Projects/Pass Random Items/ will be created if necessary, and the project file Pass Random Items.xcode will be created therein.

After you complete this step, Xcode displays the new project in its window, shown in Figure 2-3.

Figure 2-3 The files of an AppleScript action project in Xcode



File Name		Code			
AppKit.framework					☐
AppleScriptKit.sdef					☐
Automator.framework					☑
Cocoa.framework					☑
Foundation.framework					☐
Info.plist					☐
InfoPlist.strings (English)					
main.applescript	✓				☑
main.nib (English)					
Pass Random Items.action					☐

Almost all of the items in the project folder have special significance in the development process.

Frameworks
: Any action project must import the Cocoa umbrella framework, which includes the Foundation and Application Kit frameworks. It also imports the Automator framework, which defines the programmatic interface for all Automator actions. See *Automator Framework Reference* for documentation of this interface.

`main.applescript`
: The main AppleScript script file whose `on run` handler is called by Automator when the action runs in a workflow. You will write your AppleScript code in this file. An AppleScript action project can also have other ("helper") AppleScript scripts, often to manually synchronize user settings with the action internal record of those settings.

`Info.plist` and `InfoPlist.strings (English)`
: The `Info.plist` file is the information property list for the action bundle. It contains configuration information that is generally related to the bundle and more specifically related to the action. The `InfoPlist.strings` file contains English translations of items in `Info.plist` that might be displayed to the user. If your action is to be localized for languages or locales besides English, you will have to add an `InfoPlist.strings` file to the project for each additional translation.

`main.nib (English)`
: The nib file for the English version of the action. A nib file is an archive containing the view, controls, and other user-interface objects used by an executable, as well as the connections between those objects. You use the Interface Builder to create and maintain nib files. If your action is to be localized for languages or locales besides English, you will have to add a `main.nib` file for each additional localization.

The `Pass Random Items.action` item shown in the project window is the action bundle. When the action project is built, the text color of the item will change from red to black to indicate that the bundle now exists in the build directory. All Automator action bundles must have an extension of `action`.

Creating the User Interface

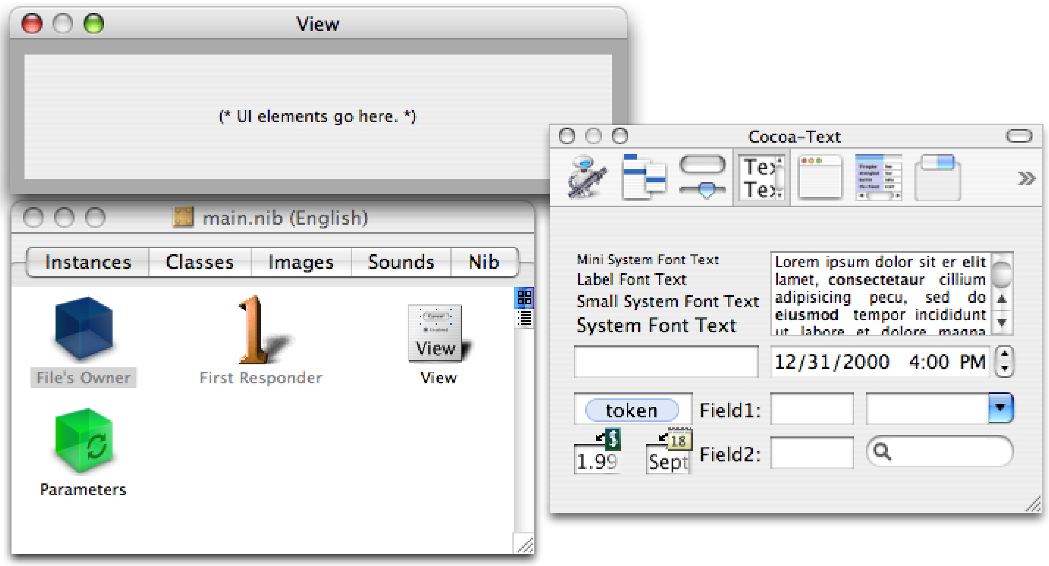
In this part of the tutorial, you will create the user interface of the Pass Random Items action. This phase of development includes not only placing, resizing, and configuring the objects of the user interface, but establishing the bindings between those objects and the parameters property of the action.

Opening the Action Nib File

The nib file is an Interface Builder archive holding the objects of a user interface and any connections between those objects. (A nib file can also contain custom class definitions and resources such as images and sounds, but the Pass Random Items action doesn't use these things, so we'll leave it at that.) The nib file for an action has the default name `main.nib`.

To open `main.nib`, double-click the icon next to the file in the project window for Pass Random Items (see Figure 2-3 (page 16). The Interface Builder launches (if it isn't running already) and displays the windows related to the nib file (see. Figure 3-1).

Figure 3-1 The nib file window, initial action view, and palette



The nib file window in this illustration is the one containing the File's Owner, First Responder, View, and Parameters icons in the Instance pane. (There are other panes for classes, images, and sounds, but you can ignore those for now.) The window with the grey rectangular area is the view of the action; here is where you will place the fields, controls, and other objects of the action view. The third window is the palette window containing palettes with various kinds of objects.

Before you start adding objects to the action view, however, make sure that the Cocoa-Automator palette is loaded. This palette contains several kinds of user-interface objects that are special to Automator. You won't need these objects for this tutorial, but you might need them for other actions. At the top of the palette window is a row of icon buttons. See if the Automator icon is one of them.

Figure 3-2 Automator icon

If the Automator icon isn't there, load the Cocoa-Automator palette:

1. Choose Preferences from the Interface Builder menu.

2. Click the Palettes button to display the list of currently loaded palettes.

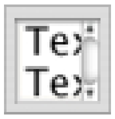
3. Click the Add button.

4. In the file browser, navigate to `/Developer/Extras/Palettes` and select the `AMPalette.palette` item.

Placing and Configuring User-Interface Objects

The user interface of the Pass Random Items action is simple, consisting of only a few text fields and one matrix object holding two radio buttons. Simplicity is one of the design principles for all actions. "Design Guidelines for Actions in the *Automator Programming Guide* discusses guidelines for action views.

Let's begin. Select the text palette by clicking the text button icon at the top of the palette window:



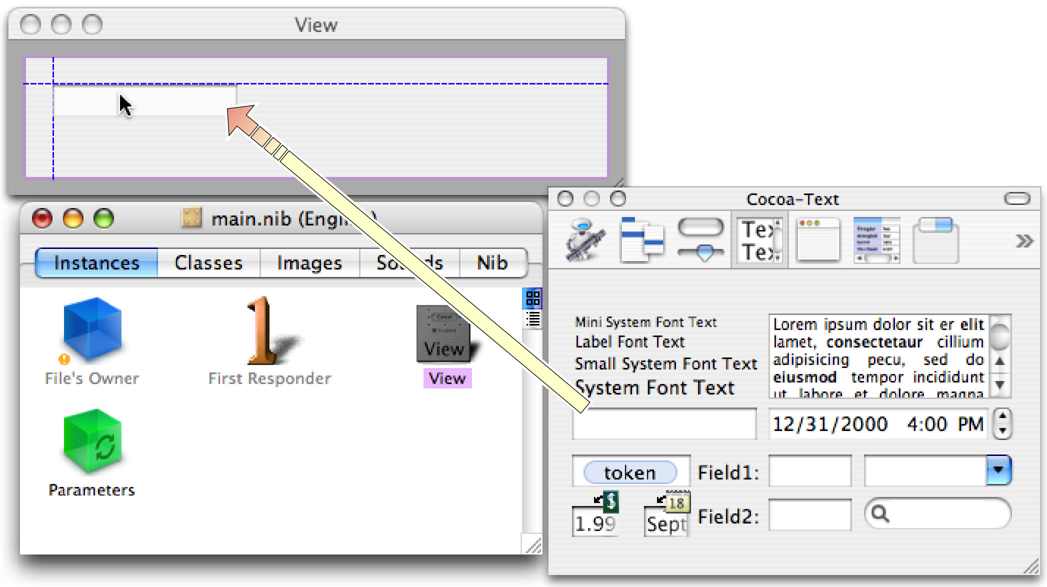
The text palette contains various types of objects related to text: editable text fields, labels, token fields, search fields, forms, and so on. First place a text field on the action view; users will enter a number or a percentage in this field, depending on the radio button selected.

1. Drag the text field from the palette to the upper-left corner of the action view.

Interface Builder uses blue lines to show you the proper location for object placement according to *Apple Human Interface Guidelines*. Make sure the top and left side of the text field are adjacent to the blue lines that appear (see Figure 3-3 (page 21)).

2. Release the mouse button to "drop" the object.

Figure 3-3 Placing a text field by dragging it from the palette



Note that after you drop an object in a view, you can still select it and move it within the view.

Many of the objects in a user interface—for example, text fields, buttons, and table columns—have three predefined sizes: mini, small, and regular (or system). Objects in an action view should always be small. The text field that you just placed is not. To change the text field to a small size, do the following;

1. Select the text field.

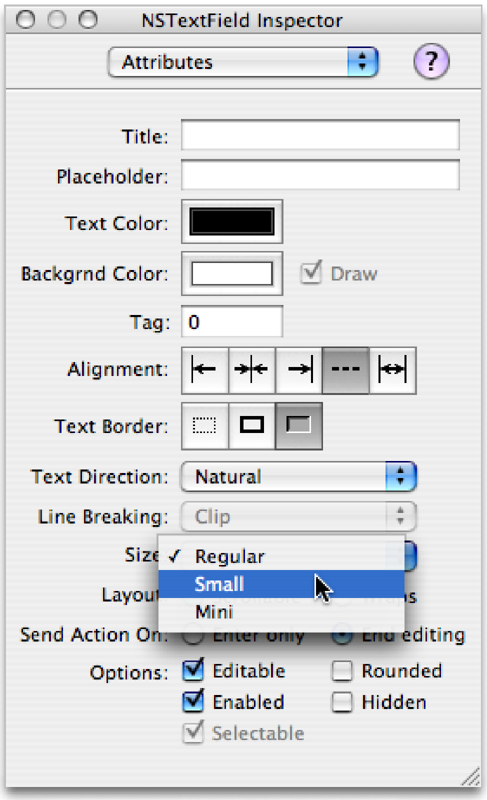
2. Choose Show Inspector from the Tools menu.

3. Select the Attributes pane from the pop-up list of the inspector.

   The Attributes pane shows all of the configurable options for whatever object is selected. For text fields, these options include color, alignment, and the fields enabled and editable states.

4. Choose Small from the Size pop-up list (see Figure 3-4).

Figure 3-4 Setting the size attribute of the text field


NSTextField Inspector

Attributes

Title:

Placeholder:

Text Color:

Backgrnd Color: ✓ Draw

Tag: 0

Alignment:

Text Border:

Text Direction: Natural

Line Breaking: Clip

Size: ✓ Regular / Small / Mini

Layout:

Send Action On: Enter only / End editing

Options: ✓ Editable, ✓ Enabled, ✓ Selectable, Rounded, Hidden

A couple of other things are not quite right with the text field. It is wider than we need for a simple number or percentage. And the field should have the label "Pass:" just to its left. We can solve these problems by resizing the text field to the right.

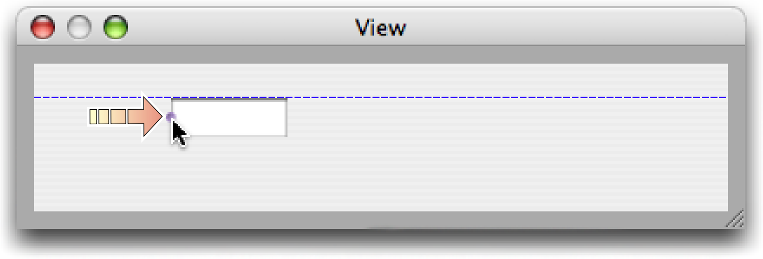
1. Select the text field.

   When it's selected, you see tiny round handles on each side and on each corner. You can use these handles to make an object larger or smaller in the given horizontal, vertical, or diagonal direction.

2. Press the mouse pointer down on the handle on the left side of the text field (not the corner handles).

3. Drag the handle toward the right until the text field is about half the original size (see Figure 3-5).

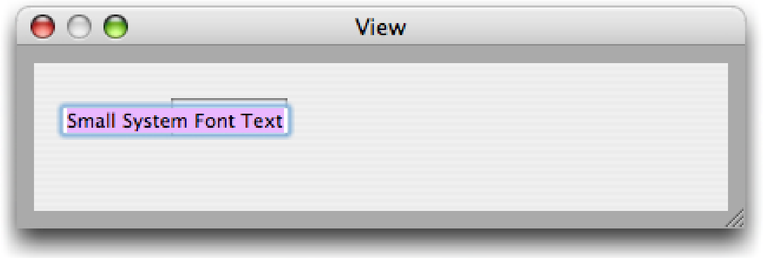
Figure 3-5 Resizing the text field



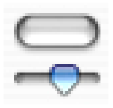
A label is a text field that has a neutral background and that is non-editable. To create the label for the text field:

1. Drag the object on the text palette that reads "Small System Font Text" and drop it in the upper-left corner of the action view.
2. Double click this generic label to select the text (see Figure 3-6).
3. Edit the text to say "Pass:".

Figure 3-6 Changing the string in a in a non-editable text field (a label)



The next step is adding the radio buttons labeled "Number" and "Percentage". Radio buttons are on the Cocoa Controls and Indicators palette. You can access this palette by clicking the following icon button at the top of the palette window:



Radio buttons are preconfigured compound objects. They are designed to work with a group of identical buttons in a way that ensures only one of them is enabled at any time. The object that holds these multiple objects together is a matrix.

1. Drag the palette object with two "Radio" buttons onto the action view just to the right of the text field.

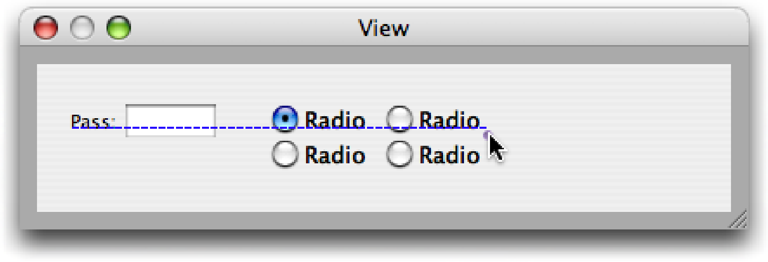
   Even though the objects are labeled "Radio", this is a matrix object.

2. With the radio-button matrix selected, press the middle handle on the right side of the matrix.
3. Option-drag the matrix to the right until two more "Radio" buttons appear (see Figure 3-7).

   "Option-drag" means to press the option key while dragging the object handle.
4. Select the lower-middle handle of the matrix.
5. Option-drag the matrix upward until the two bottom "Radio" buttons disappear.

Now there are two buttons on the same row.

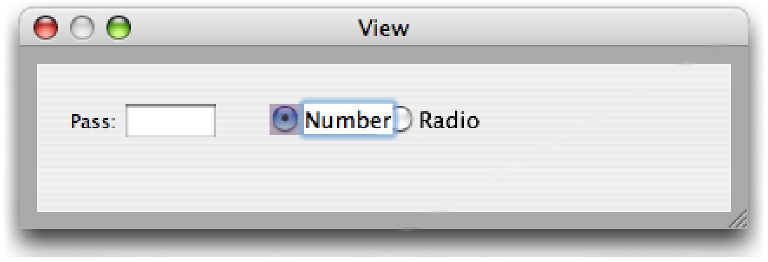
Figure 3-7 Changing the number of radio buttons in a matrix (radio buttons)



The user interface is looking much better, but you still have some work to do. The buttons are too large, and they need the correct titles. Fortunately, you can solve both of these problems at the same time for each button:

1. Double-click a radio button (a cell) to select its text.
2. Change the text to "Number" or "Percentage" (see Figure 3-8).
3. In the Attributes pane of the inspector for the button cell, change the size to Small.

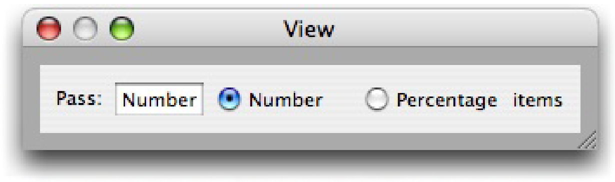
Figure 3-8 Changing the title of a button



The user interface of the Pass Random Items action needs one final object. Add a small label after the "Percentage" radio button that reads "items".

But you're not finished yet. The action view is too big for the objects it contains. To resize the view, click and press the lower-right corner of the view window, then move the window in toward the upper left corner of the view until all objects are just contained. Make sure that the objects on the view conform to the blue guidance lines. Then add back a 10-pixel border around the user-interface objects on all sides; this border is required by the user-interface guidelines for actions. The final action view should look like the example in Figure 3-9.

Figure 3-9 Final user interface of Pass Random Items action



Establishing Bindings

The objects on an action view are only part of what's involved in creating the user interface of an action. If users click a button or enter something into a text field, nothing much happens until you communicate those events to other objects in the action that know how to deal with the events. Even though you are creating an AppleScript action, the underlying framework of Automator is Cocoa-based. Cocoa gives you two general mechanisms for enabling communication between view objects and other objects in a action:

- Outlets and target-action ("action" here does not denote an Automator action)
- Bindings

The preferred approach for managing an action's user interface is to use the Cocoa bindings technology; that is how actions projects are initially configured in the project templates and that is the procedure this tutorial shows. But you can manage the user interface using an alternate approach that makes use of outlets and possibly target-action. "Alternatives to Bindings" (page 31) summarizes this approach.

Note: The technology of Cocoa bindings relies on a number of APIs and mechanisms that this tutorial won't go into. If you are interested in learning about them, read *Cocoa Bindings Programming Topics*.

Bindings in an Action

A binding in Cocoa automatically synchronizes the value between an attribute of a user-interface object (say, the displayed value of a text field) and a property of a data-bearing object (usually termed a model object). This means that whenever a user edits a control or clicks a button, that change is automatically communicated to the bound property of the object maintaining that value internally. And whenever that internal value changes, the change is automatically communicated to the bound attribute of a user-interface object that then displays it.

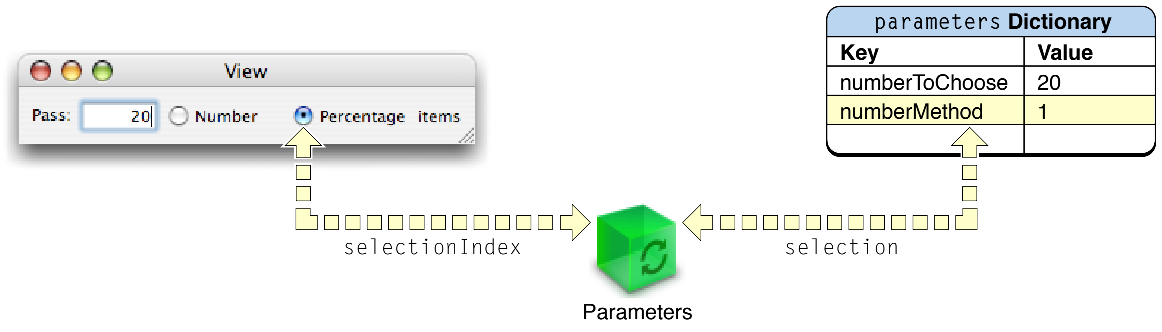
Note: This part of the tutorial frequently talks about the properties of objects. "Property" in this sense means an essential characteristic of the object that it encapsulates. A property can either be an attribute, such as a color or a person's name, or a relationship—that is, a reference to one or more other objects. In Cocoa bindings, the values of properties are accessed using their names as keys.

For actions the data-bearing object is a dictionary owned by the action object itself. For AppleScript actions, the action object is almost always an instance of AMAppleScriptAction. Every action, regardless of the programming or scripting language it uses, maintains an internal dictionary that captures the choices users have made in the user interface. (The AppleScript equivalent for a dictionary is a record.) This dictionary is called the parameters dictionary. It stores values users make in the user interface along with an arbitrary key for each value. When Automator runs an AppleScript action in a workflow, it passes it a `parameters` record in the `on run` handler in `main.applescript` (See "Writing the Action Script" (page 41) for more about the `on run` handler.)

When you establish a binding between a user-interface control and a property of the action's parameters dictionary, the binding is made through a property of an intermediary object called a controller. In the `main.nib` file for an action, this intermediary object appears in the Instance view of the nib file window as the Parameters instance. When you look at a binding in Interface Builder in the Bindings pane of the inspector (see Figure 4-4 (page 31) for an example), you can see it as a combination of user-interface attribute, controller property, and parameters property.

Figure 4-1 illustrates the case of the radio-button matrix of the Pass Random Items action; here the matrix attribute `selectionIndex` is connected to the controller's `selection` property, which is connected to the `numberMethod` property of the parameters dictionary. The value of `numberMethod` reflects the zero-based index of the selected radio button in the matrix (1 indicates the "Percentage" button in the example).

Figure 4-1 Binding between the pop-up list and a property of the parameter dictionary



`parameters` **Dictionary**

Key	Value
numberToChoose	20
numberMethod	1

Establishing the Bindings of the Action

To establish bindings for the Pass Random Items action, complete the following steps with the action's `main.nib` file open in Interface Builder:

1. Select the Parameters instance in the nib file window.

   Parameters is an instance of NSObjectController, which implements controller behavior.

2. Open the inspector window (Tools > Show Inspector) and choose Attributes from the pop-up list.

3. In the Attributes pane for the Parameters instance, click Add.

   A `newKey` placeholder appears in the Keys table.

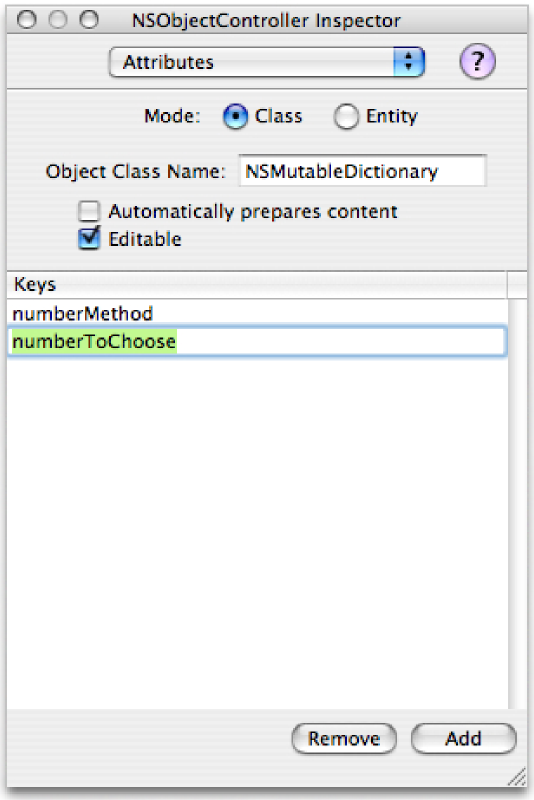
4. Double-click `newKey` to select the word and make it editable.

5. Type `numberMethod`, replacing `newKey`.

6. Click Add again, and add another key named `numberToChoose`.

   See Figure 4-2 for an example.

Figure 4-2 Adding keys as attributes of the Parameters instance.



NSObjectController Inspector

Attributes

Mode: [✓ Class] Entity

Object Class Name: NSMutableDictionary

☐ Automatically prepares content
☑ Editable

Keys
numberMethod
numberToChoose

The Parameters controller instance is now initialized with the keys that will be used in the bindings between attributes of two of the user-interface objects and properties of the parameters dictionary. Note that the project template for all types of actions is preconfigured to make a binding between the Parameters instance and the action's parameters dictionary. To see this binding:

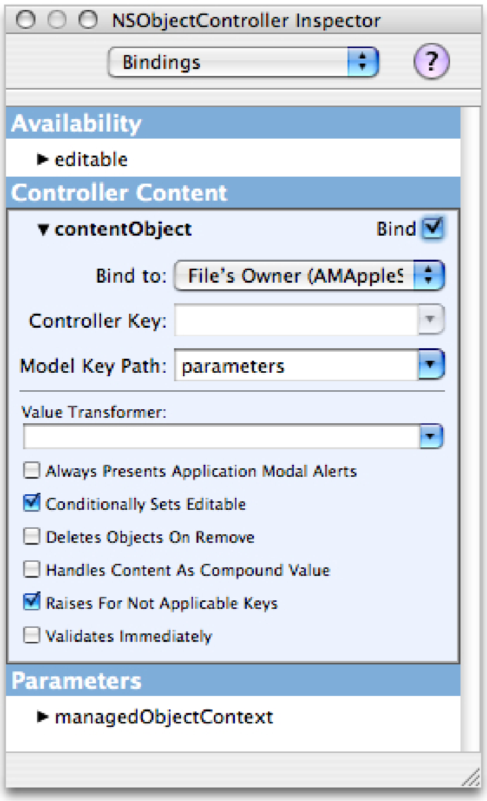
1. Select the Parameters instance in the nib file window.

2. Choose Bindings from the inspector's pop-up list.

3. Click the disclosure triangle next to `contentObject` to expand the view.

   Figure 4-3 shows the binding between the controller object and the `parameters` property of the action object (File's Owner).

Figure 4-3 Binding between the controller and the parameters dictionary



Bind to: File's Owner (AMAppleS
Controller Key:
Model Key Path: parameters
Value Transformer:
☐ Always Presents Application Modal Alerts
☑ Conditionally Sets Editable
☐ Deletes Objects On Remove
☐ Handles Content As Compound Value
☑ Raises For Not Applicable Keys
☐ Validates Immediately

The final stage of establishing bindings requires you to bind the attributes of two of the user-interface objects to the corresponding properties of the parameters dictionary via the `selection` property of the Parameters controller.

1. Select the radio-button matrix in the action view.

2. Choose Bindings from the inspector's pop-up list.

3. Click the disclosure triangle next to the `selectedIndex` attribute of the matrix.

4. Make sure the Bind to pop-up list is set to Parameters.

5. Make sure the Controller Key combo box is set to `selection`.

6. Set the value of the Model Key Path combo box to `numberMethod`.

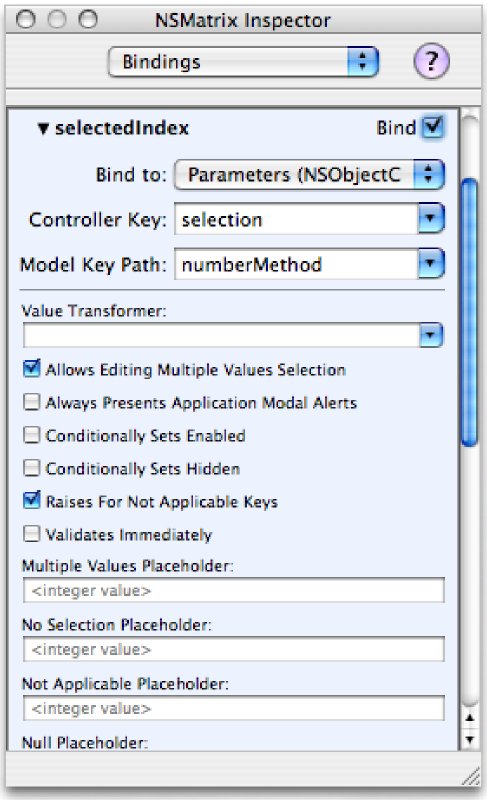
7. Make sure the Bind check box in the upper-right corner of the `selectedIndex` view is checked.

   The Bindings inspector pane should look like the example in Figure 4-4 (page 31) at this point.

8. Select the text field to the left of the matrix.
9. In the Bindings pane of the inspector, click the disclosure triangle next to the value attribute.
10. Make sure the Bind to combo box contains Parameters and the Controller Key combo box contains `selection`.
11. Set the Model Key Path combo box to `numberToChoose`.
12. Make sure the Continuously Updates Value check box is checked.

    Checking this control tells the bindings mechanism to synchronize the value in the text field without waiting for the user to press the Return or Tab keys.
13. Make sure the Bind check box is checked.

Figure 4-4 The selectedIndex attribute in the bindings inspector



▼ selectedIndex — Bind ✓
Bind to: Parameters (NSObjectC
Controller Key: selection
Model Key Path: numberMethod
Value Transformer:
✓ Allows Editing Multiple Values Selection
☐ Always Presents Application Modal Alerts
☐ Conditionally Sets Enabled
☐ Conditionally Sets Hidden
✓ Raises For Not Applicable Keys
☐ Validates Immediately
Multiple Values Placeholder: <integer value>
No Selection Placeholder: <integer value>
Not Applicable Placeholder: <integer value>
Null Placeholder:

Alternatives to Bindings

Although bindings are the preferred technique for enabling communication the objects of an action, there are alternatives to bindings. For example, you can use outlets and target-action to facilitate the communication of data between objects in the action view and the parameters dictionary owned by the action object. In this case you also use a controller object, but instead of bindings it maintains persistent references to other objects known as outlets. Thus it can always send a message to, say, a text field to obtain its value. User-interface objects and controllers can also be set up to use target-action. In this mechanism a control object (such as a button) is configured with a target of a message—usually the controller—and a selector that designates the message to send. When users activate the control object, a message is automatically sent to the controller. You can establish outlet and target-action connections in Interface Builder, which archives these connections in the nib file.

Note: To learn more about outlets and target-action, see *Cocoa Fundamentals Guide*.

Automator provides a third alternative for synchronizing the values in the parameters and the settings users make in the action's user interface. It defines the `update parameters` and `parameters updated` commands, which you can attach to an action's view using AppleScript Studio. Automator sends the `update parameters` command when an action's parameters need to be refreshed from the values on the user interface. It sends `parameters updated` when there are any changes to the action's parameters dictionary. "Implementing an AppleScript Action" in *Automator Programming Guide* describes this procedure in detail.

Configuring the Action

Every bundle in Mac OS X—and that includes applications, frameworks, and loadable bundles such as actions—has an information property list. This property list, which is contained in a file named `Info.plist`, is a series of key-value pairs in XML format. The standard information property list defines such properties of the bundle as its identifier, its type code, and its main class.

Information property lists can contain properties other than the standard ones. Such is the case with Automator. In the `Info.plist` file of an action project you can (and in some cases must) specify properties that characterize the action, enabling Automator to display information about the action and handle it properly. For example, some Automator properties provide the name and description of an action and others indicate what types of data the action operates on (or produces). The following sections describe the basic approach to completing the action-specific properties and steps you through the properties that you must specify for Pass Random Items.

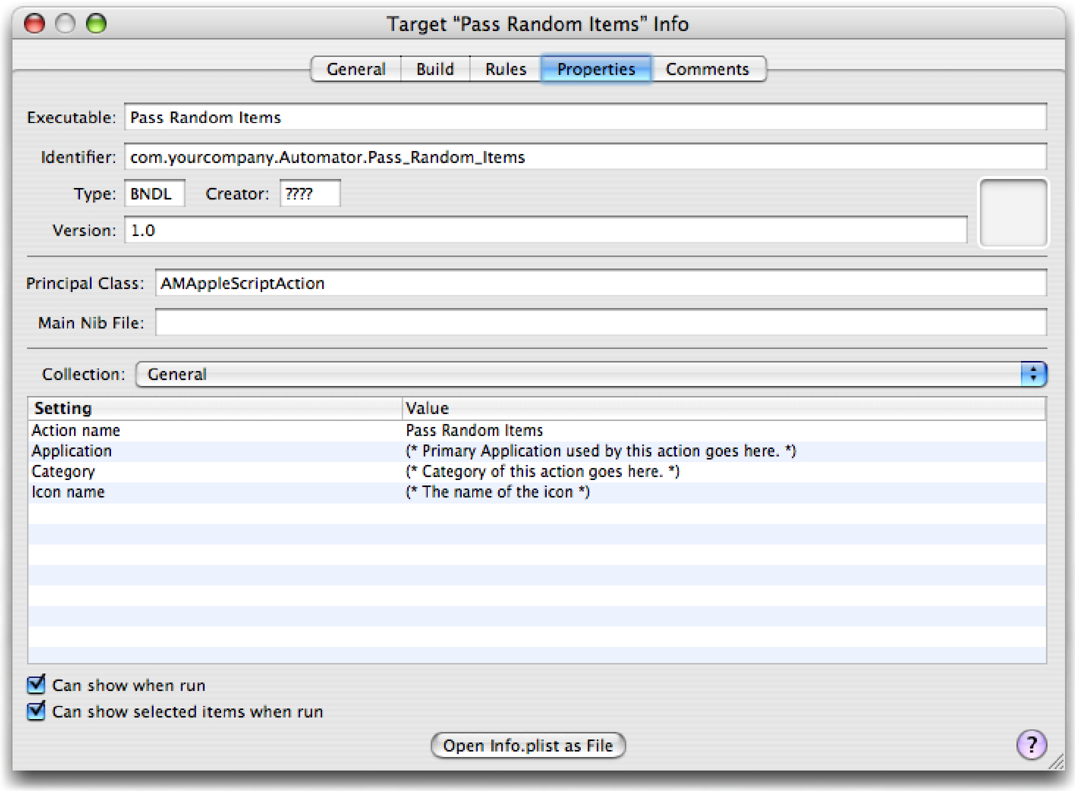
For complete descriptions of the Automator properties, see “Automator Action Property Reference” in *Automator Programming Guide*.

Note: The inspector for Automator properties was first introduced in Xcode 2.1. If you have an earlier version of Xcode, you have to edit the properties in the `Info.plist` file manually.

Editing the Information Property List

Automator action projects take advantage of a special inspector built into Xcode for viewing and editing the action’s information property list. To access this editor, choose Edit Active Target ‘Pass Random Items’ from the Project menu. Then, click the Properties tab in the Target Info window to display the property inspector, which is shown in Figure 5-1.

Figure 5-1 The property inspector for Automator actions (general collection)



Executable: Pass Random Items
Identifier: com.yourcompany.Automator.Pass_Random_Items
Type: BNDL Creator: ????
Version: 1.0
Principal Class: AMAppleScriptAction
Main Nib File:
Collection: General

Setting	Value
Action name	Pass Random Items
Application	(* Primary Application used by this action goes here. *)
Category	(* Category of this action goes here. *)
Icon name	(* The name of the icon *)

✓ Can show when run
✓ Can show selected items when run

The first thing to note about the property inspector for actions is that it is divided into two parts. The upper half of the window contains general bundle properties, such as the name of the executable, the bundle identifier, and the principal class. You shouldn't have to change any of these values.

The lower half of the window is the area for viewing and editing Automator-specific properties. The Collection pop-up list displays various groupings of properties. The first displayed is the General group. Note that the action name (and the last part of the bundle identifier) are automatically assigned the name of the Xcode project. (This automatic name assignment was introduced in Xcode 2.1.) You are going to keep the name for the action, but assign values to the Application, Category, and Icon name properties.

Note: In the `Info.plist` file the property keys are different from the strings displayed in the inspector. The keys have "AM" prefixes and no spaces between words, for example, `AMActionName`, `AMApplication`, `AMCategory`, and `AMIconName`.

1. Double-click the cell under Value containing the comment for the Application property. This selects the cell and makes it editable.

2. Replace the comment with "Automator".

   The application named here is either the one that the action primarily sends scripting commands to or the application that the action is most closely associated with.

3. Replace the comment for Category with "Utility".

Automator uses an action's category in searches, along with keywords.

4. Replace the comment for Icon name with "Action".

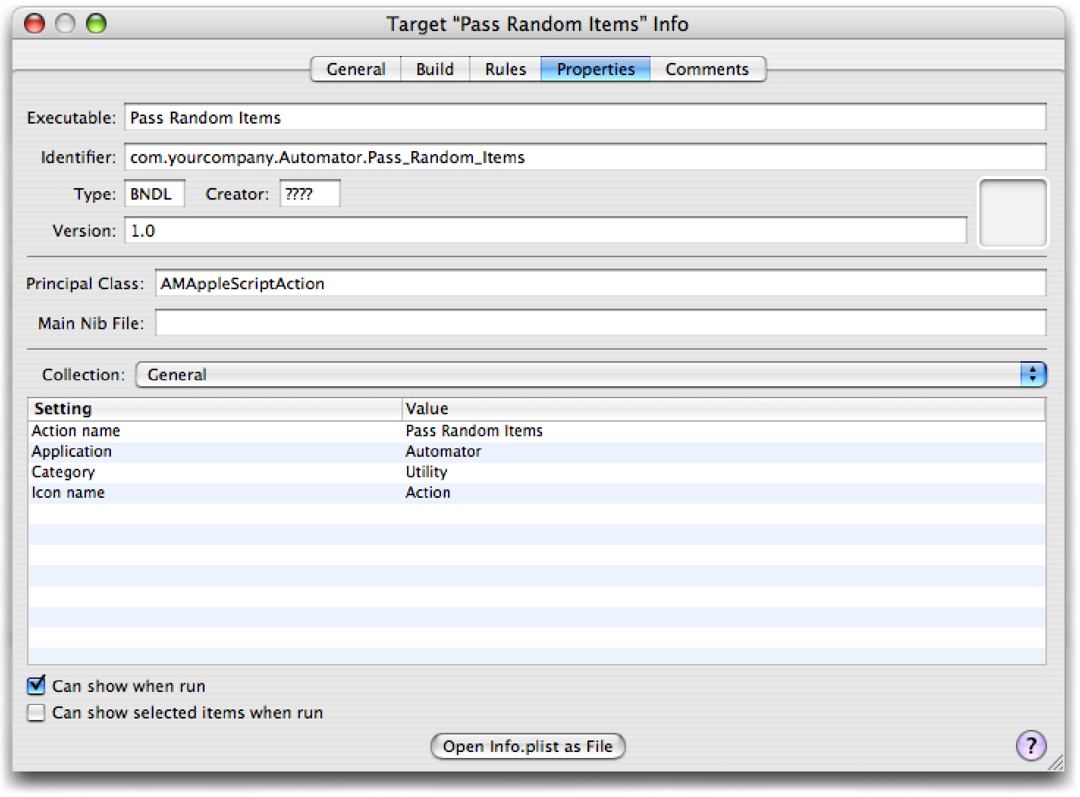
   This value requests a generic icon for actions to be displayed next to the action name in Automator.

5. Uncheck the check box labeled "Can show selected items when run." Leave the check box "Can show when run" checked.

   This pair of settings allows you to specify values in the action's user interface when the workflow containing it is executed; the entire user interface is displayed, not a subset of it. For more on the show-when-run feature, see "Show When Run" in *Automator Programming Guide*.

When you have finished these steps, the General collection of properties should look like the example in Figure 5-2.

Figure 5-2 The completed General collection of Automator properties)



Executable: Pass Random Items
Identifier: com.yourcompany.Automator.Pass_Random_Items
Type: BNDL Creator: ????
Version: 1.0
Principal Class: AMAppleScriptAction
Main Nib File:
Collection: General

Setting	Value
Action name	Pass Random Items
Application	Automator
Category	Utility
Icon name	Action

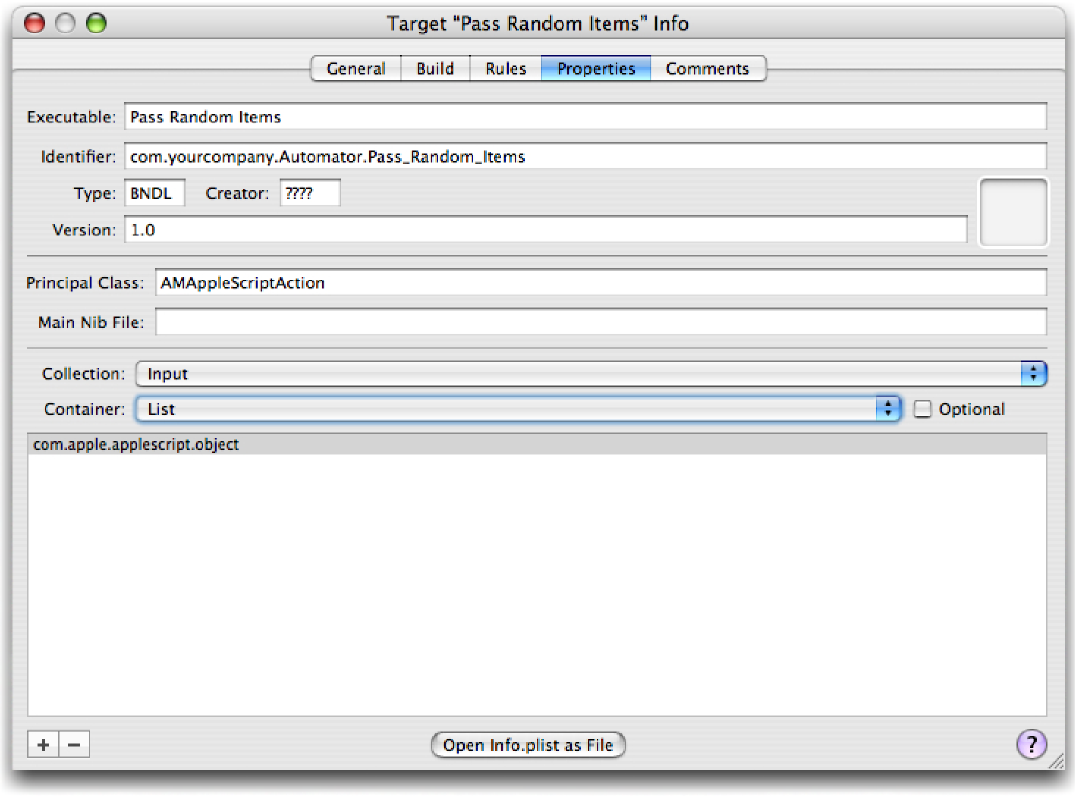
[✓] Can show when run
[ ] Can show selected items when run

Action Input and Output

Every action must specify what types of data it accepts from the action before it in the workflow and what types of data it provides to the next action in the workflow. The `AMAccepts` and `AMProvides` properties of the information property list allow you to do this. The Automator property inspector of Xcode shows these properties as the Input and Output collections.

Choose the Input collection from the pop-up list to display the view of the inspector shown in Figure 5-3.

Figure 5-3 Automator property inspector—Input collection



Executable: Pass Random Items
Identifier: com.yourcompany.Automator.Pass_Random_Items
Type: BNDL Creator: ????
Version: 1.0
Principal Class: AMAppleScriptAction
Main Nib File:
Collection: Input
Container: List ☐ Optional
com.apple.applescript.object

The main view for the Input collection is a single-column, headingless table. You click the plus (+) button to add a cell for a new entry and click the minus button (-) to delete the selected entry. The entries in this table are UTI-style type identifiers specifying the types of data the action can accept. The `Types` subproperty of the `AMAccepts` property for AppleScript action projects is initialized to `com.apple.applescript.object`, which means that the action can accept any type of AppleScript object as input. Since this fits the type of data that the Pass Random Items action can process—it merely passes on a random subset of the items passed it—you do not need to modify contents of the table.

When you are developing your own actions, you will probably want to specify different types of identifiers. For example, if your action handles iTunes songs, you would specify `com.apple.itunes.track-object`. If your action can handle URLs, you would enter `public.url` in the Input table. It's best to be as specific as possible about the types of data that your action can accept and provide. For a listing of supported type identifiers for actions, see "Automator Action Property Reference" in *Automator Programming Guide*.

The Input collection part of the inspector has two additional controls: a Container pop-up list and an Optional check box. These controls correspond to two subproperties of `AMAccepts`: `Container` and `Optional`. The former indicates whether the input data is a single item or a list of items; this control is almost always left as List. The latter control indicates whether input is optional for the action. Leave both controls unchanged.

The settings of the Output collection for Pass Random Items are identical to those for the Input collection. The type identifier is `com.apple.applescript.object` and Container is set to List.

Default Parameter Settings

The `AMDefaultParameters` property allows you to specify initial values for action controls when the action first appears in a workflow. In the Automator property inspector, you edit this property through the Parameters collection. The table for this property has three columns:

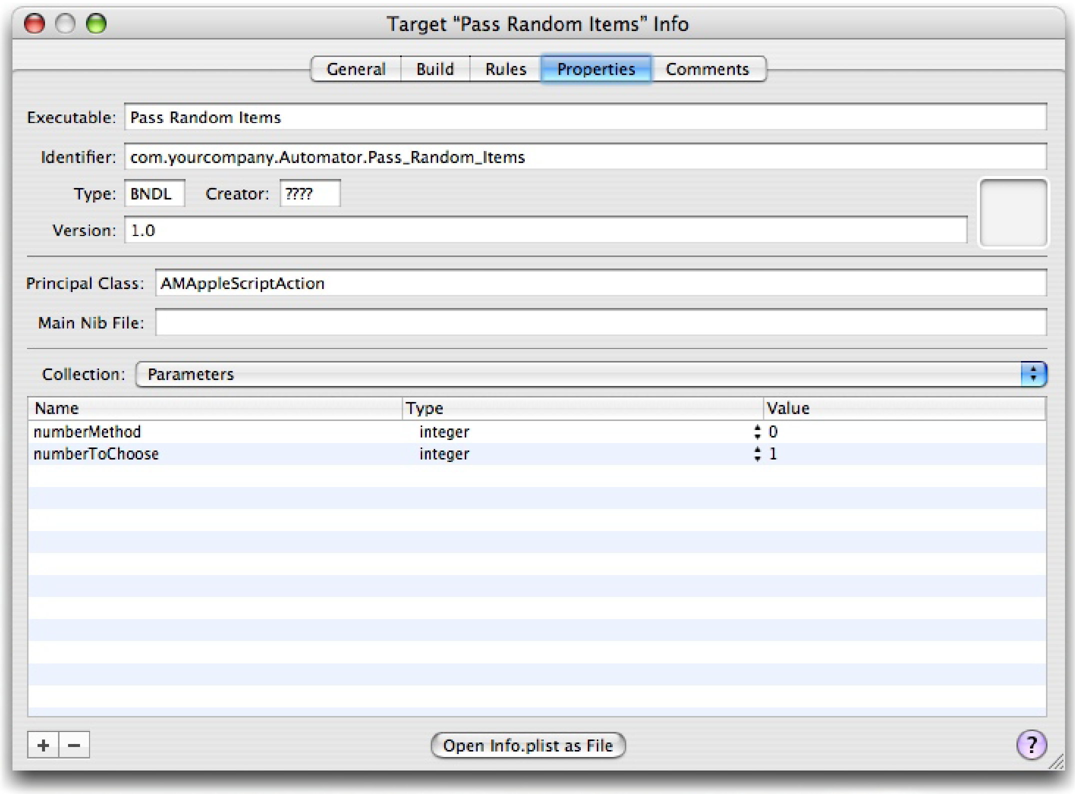
- Name — the key identifying the control and its associated property in the parameters dictionary
- Type — the type of data represented by the control
- Value — the initial value for the control

To set the initial value for the text field containing the number or percentage, do the following:

1. Click the plus button (+) below the table to insert a new row into the table.
2. Double-click the cell under the Name column to open it for editing.
3. Type “numberToChoose” in the cell; this is the same name that you gave the binding key in Interface Builder.
4. In the Type column of the same row, select “integer” from the pop-up list attached to the cell.
5. Double-click the cell under Value in the same row and type “1”.

Repeat the same procedure for the radio-button matrix, entering “numberMethod” for the name, “integer” for the type, and “0” for the value. When you are finished, the inspector window should look like the example in Figure 5-4.

Figure 5-4 The default parameters for the Pass Random Items action



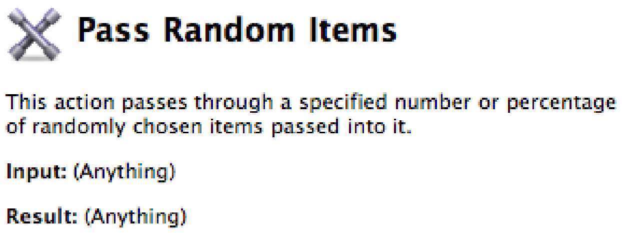
Executable: Pass Random Items
Identifier: com.yourcompany.Automator.Pass_Random_Items
Type: BNDL Creator: ????
Version: 1.0
Principal Class: AMAppleScriptAction
Main Nib File:
Collection: Parameters

Name	Type	Value
numberMethod	integer	0
numberToChoose	integer	1

The Action Description

When users browse through the actions available for Automator, they can see a brief description for each selected action in a small view in the lower-left corner of the application. When users select the Pass Random Items action, you want them to see the description shown in Figure 5-5.

Figure 5-5 The description for the Pass Random Items action



Pass Random Items

This action passes through a specified number or percentage of randomly chosen items passed into it.

Input: (Anything)

Result: (Anything)

Descriptions can be simple like this one, or they can include things like requirements and warnings. The Automator property for descriptions is `AMDescription`; it has several subproperties for the various components of description: `AMDAlert`, `AMDInput`, `AMDNote`, `AMDOptions`, `AMDRelatedActions`, `AMDResult`, and `AMDSummary`.

For the Pass Random Items action you need only set one of these subproperties, `AMDSummary`, which is the short description of what the action does. The icon, title, Input, and Result parts of the description are automatically supplied by, respectively, the `AMIconName`, `AMActionName`, `AMAccepts`, and `AMProvides` properties. (You can provide your own description for the latter two properties in addition to the defaults, if you want.)

To set the `AMDSummary` property, do the following:

1. Choose the Description collection in the property inspector.
2. Double-click the cell under the Value column for the Summary row.
3. Replace the comment with the sentence shown in Figure 5-5 (page 38).

Other Settings

There are several other Automator properties which you can access through the Collection pop-up list. One very useful property to supply values for is the Keywords property (`AMKeywords`). The Keywords collection part of the inspector provides a simple table to which you can add a list of words that identify the action in Automator searches for actions. Other collections are Required Resources (`AMRequiredResources`), Warning (`AMWarning`), and Related Actions (`AMRelatedActions`). Read "Automator Action Property Reference" in *Automator Programming Guide* to learn more about these properties.

Writing the Action Script

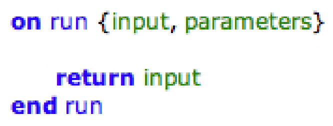
The next stage of developing the Pass Random Items action is writing the script itself. This chapter describes how to write the command handler that all AppleScript actions must implement and discusses subroutines and other aspects of scripting for actions.

For more information on this subject, see "Implementing an AppleScript Action" in *Automator Programming Guide*.

The on run Command Handler

In the Xcode project window for the Pass Random Items action project, locate the `main.applescript` file and double-click it. The file opens in an editor much like Script Editor. It contains a "skeleton" `on run` command handler, as shown in Figure 6-1.

Figure 6-1 The template for the `on run` handler



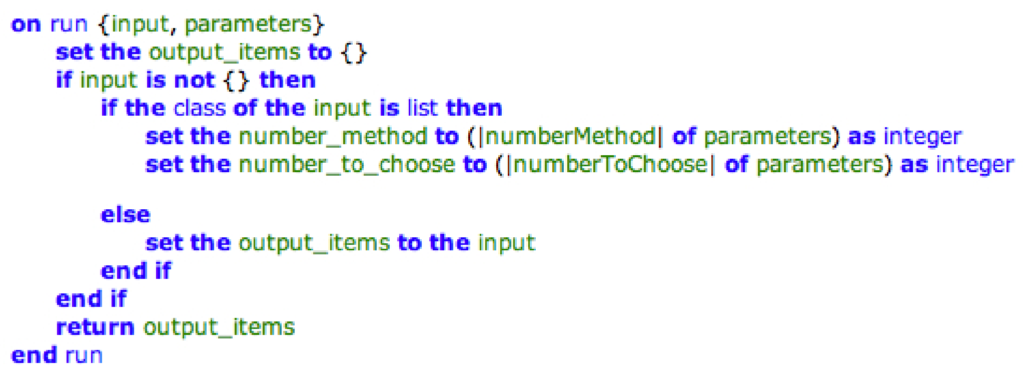
```
on run {input, parameters}

    return input
end run
```

Let's briefly look at this command handler before writing anything. Automator invokes the handler when it is an action's turn in a workflow to run. The handler has two parameters: `input` and `parameters`. The `input` parameter is the data provided by the previous action in the workflow. The template `on run` handler simply returns the input as its output. The `parameters` parameter is a record that contains the settings users have made in the action's view.

Start by initializing a list of items to return as output and extracting the settings users have made from the parameters record. Figure 6-2 shows you the scripting code to write.

Figure 6-2 Initializing local output and parameter variables



```
on run {input, parameters}
    set the output_items to {}
    if input is not {} then
        if the class of the input is list then
            set the number_method to (|numberMethod| of parameters) as integer
            set the number_to_choose to (|numberToChoose| of parameters) as integer

        else
            set the output_items to the input
        end if
    end if
    return output_items
end run
```

The first line initializes a list named `output_items` and the last line returns this list. In between, the script tests whether the input object is an empty list or is a single item instead of list and returns that as output (if a single item, it adds it to the `output_items` list first).

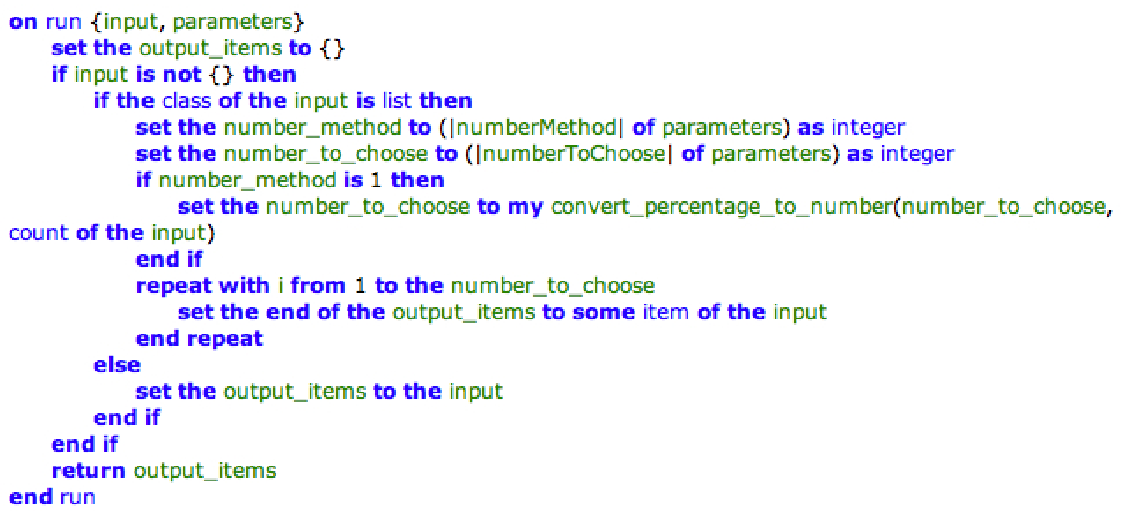
The other lines of the script in Figure 6-2 assign to local variables the values in the parameters record that are bound to the action's user-interface controls. Note that in the expression

```
(|numberToChoose| of parameters)
```

that `numberToChoose` is one of the keys you added to the attributes of the Parameters instance in Interface Builder when you established the bindings of the action. In the script you are using this key to access the value corresponding to the choice the user made in the user interface.

Finally, add the remaining lines shown in Figure 6-3 to complete the `on run` command handler.

Figure 6-3 The final `on run` handler



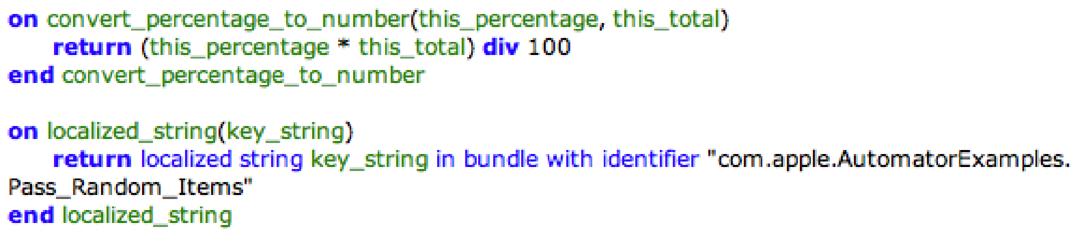
```
on run {input, parameters}
    set the output_items to {}
    if input is not {} then
        if the class of the input is list then
            set the number_method to (|numberMethod| of parameters) as integer
            set the number_to_choose to (|numberToChoose| of parameters) as integer
            if number_method is 1 then
                set the number_to_choose to my convert_percentage_to_number(number_to_choose,
count of the input)
            end if
            repeat with i from 1 to the number_to_choose
                set the end of the output_items to some item of the input
            end repeat
        else
            set the output_items to the input
        end if
    end if
    return output_items
end run
```

These lines of the script test whether the user selected the Number or Percentage radio button in the user interface; if it is Percentage, the script calls a subroutine to get the specified percentage of the input items as a number. Then in a loop it adds a random selection of input items—limited by the specified or computed number—to the output items.

Writing the Subroutines

The `main.applescript` file for the Pass Random Items action includes two subroutines. The first, `convert_percentage_to_number`, you have already encountered when writing the `on run` handler script. This subroutine performs the simple calculation shown in Figure 6-4.

Figure 6-4 Subroutines called by the main script



```
on convert_percentage_to_number(this_percentage, this_total)
    return (this_percentage * this_total) div 100
end convert_percentage_to_number

on localized_string(key_string)
    return localized string key_string in bundle with identifier "com.apple.AutomatorExamples.
Pass_Random_Items"
end localized_string
```

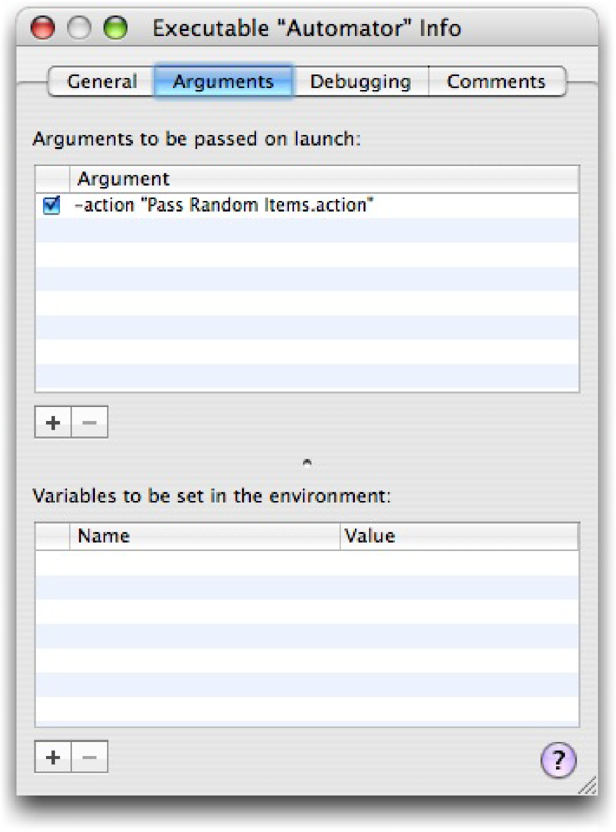
The second subroutine, `localized_string`, does something very important despite the fact that it's not called by the `on run` command handler you have written. Through the `localized string` command, the subroutine returns a string (identified by `key_string`) for a preferred localization specified by the current user in System Preferences. You can use this string in dialogs and error messages. To use this subroutine effectively you must first internationalize your action for all supported localizations. To find out how to do this, see the relevant section in Developing Actions of the *Automator Programming Guide*.

Building and Testing the Action

You have completed the steps required to develop the Pass Random Items action. You've created the user interface, established bindings, specified the `Info.plist` properties, and written the script. It's time to build and test the action.

But before you begin, look at how an action project sets up its executable for testing. Choose Edit Active Executable 'Automator' from the Project menu to display the Executable Info window. In the General pane of this window, you can see that the executable path is initialized to /Applications/Automator.app. Then click the tab for the Arguments pane; as Figure 7-1 shows, the `-action` argument passed to Automator tells it to load the Pass Random Items action.

Figure 7-1 Executable settings for the Pass Random Items action



Arguments to be passed on launch:

	Argument
✓	-action "Pass Random Items.action"

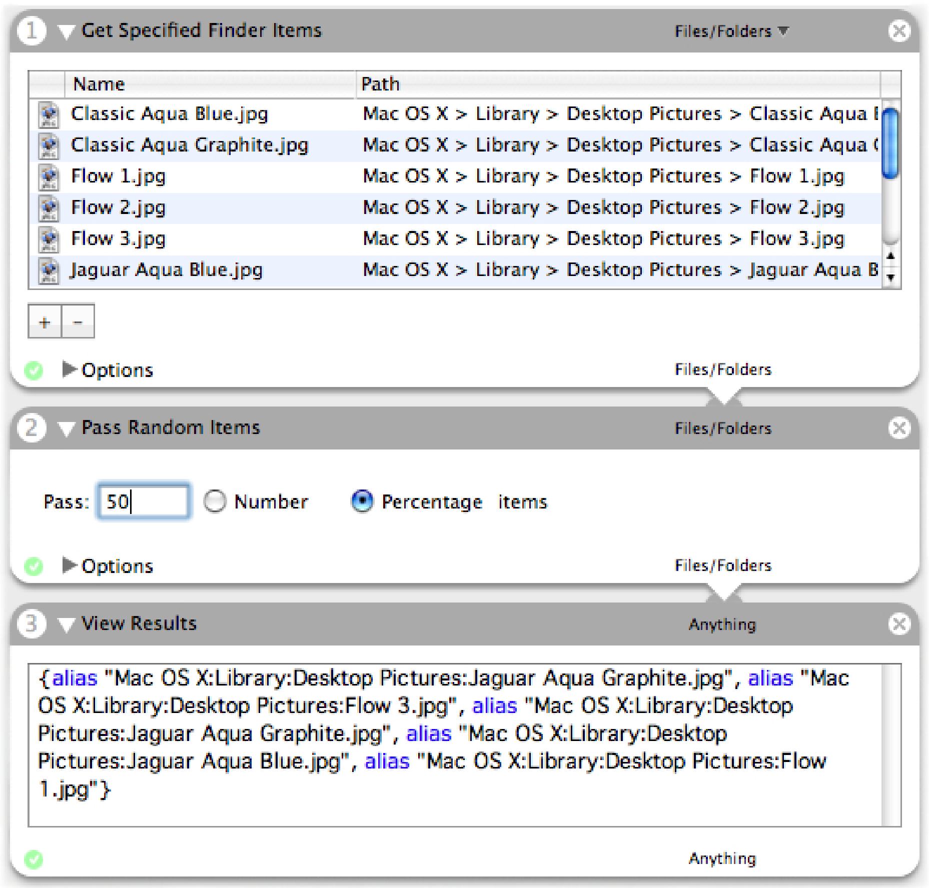
Variables to be set in the environment:

Name	Value

To build and test the action in Xcode, choose Build and Run from the Build menu. Xcode builds the action and then launches Automator. As part of the build process, Xcode runs the `amlint` utility to perform a number of action-specific tests. The results of these tests appear along with all other build results.

Assuming the action builds without error or warning and Automator launches, the next thing you should do is compose a workflow in which users would likely include the Pass Random Items action. Figure 7-2 shows a possible workflow. The Get Specified Finder Items actions allows you to select a collection of Finder items and then passes it to the Pass Random Items action. You can view the output of your action using the View Results action. Check to see if the correct number or percentage was passed and if the selection is truly random.

Figure 7-2 Testing the Pass Random Items action in a workflow



1 Get Specified Finder Items — Files/Folders

Name	Path
Classic Aqua Blue.jpg	Mac OS X > Library > Desktop Pictures > Classic Aqua
Classic Aqua Graphite.jpg	Mac OS X > Library > Desktop Pictures > Classic Aqua
Flow 1.jpg	Mac OS X > Library > Desktop Pictures > Flow 1.jpg
Flow 2.jpg	Mac OS X > Library > Desktop Pictures > Flow 2.jpg
Flow 3.jpg	Mac OS X > Library > Desktop Pictures > Flow 3.jpg
Jaguar Aqua Blue.jpg	Mac OS X > Library > Desktop Pictures > Jaguar Aqua B

Options — Files/Folders

2 Pass Random Items — Files/Folders

Pass: 50 Number [✓ Percentage] items

Options — Files/Folders

3 View Results — Anything

```
{alias "Mac OS X:Library:Desktop Pictures:Jaguar Aqua Graphite.jpg", alias "Mac
OS X:Library:Desktop Pictures:Flow 3.jpg", alias "Mac OS X:Library:Desktop
Pictures:Jaguar Aqua Graphite.jpg", alias "Mac OS X:Library:Desktop
Pictures:Jaguar Aqua Blue.jpg", alias "Mac OS X:Library:Desktop Pictures:Flow
1.jpg"}
```

Anything

Automator has its own set of actions that are useful in testing. To see them, disclose the Applications folder under the Library column of the application and select Automator. View Results is one of these actions. Others that you might find useful in action development and testing are the following:

- Run AppleScript — Enables you to prototype a script before using it in an action.
- Wait For User Action — Displays a message informing users what must be done at this point for the workflow; if the action isn't completed by a specified period, it stops the workflow.
- Confirmation Dialog — Allows you to pause or cancel execution of the workflow.

If Xcode displays errors and warnings when you attempt to build the action, or if the action doesn't behave as expected, and you cannot readily pinpoint the cause of the problem, you can either debug the action (using a special AppleScript debugger) or add log statements. To debug an AppleScript action:

1. In the Xcode script editor, set a breakpoint in the script.

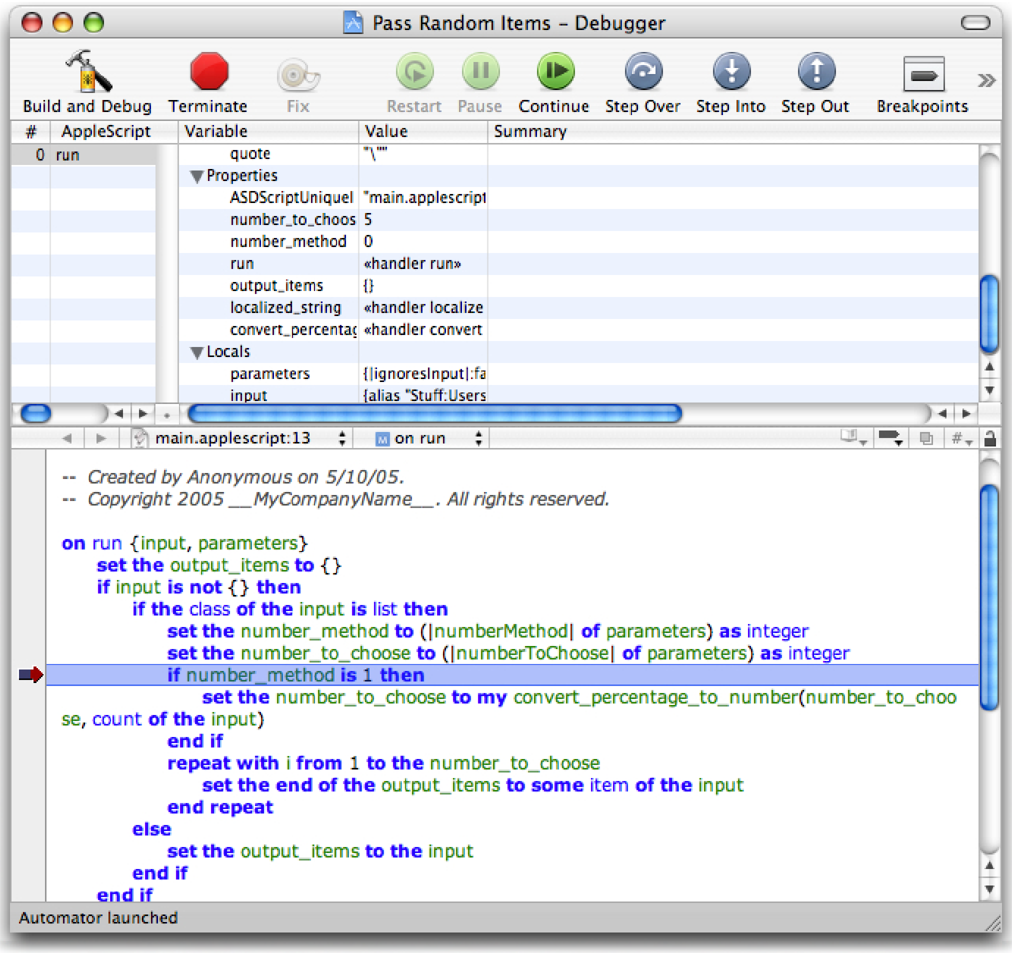
   Click in the gray strip next to the line you want the debugger to break on. A black breakpoint indicator appears in the gray strip.

2. Choose Build and Debug from the Build menu.

3. When Automator launches, construct a workflow with your action in it and execute it.

   When your action runs, the Xcode AppleScript debugger shows a debugging window similar to the one in Figure 7-3.

Figure 7-3 The AppleScript debugger



#	AppleScript
0	run

Variable	Value	Summary
quote	"\""	
▼Properties		
ASDScriptUniqueI	"main.applescript	
number_to_choos	5	
number_method	0	
run	«handler run»	
output_items	{}	
localized_string	«handler localize	
convert_percentag	«handler convert	
▼Locals		
parameters	{|ignoresInput|:fa	
input	{alias "Stuff:Users	

```
-- Created by Anonymous on 5/10/05.
-- Copyright 2005 __MyCompanyName__. All rights reserved.

on run {input, parameters}
    set the output_items to {}
    if input is not {} then
        if the class of the input is list then
            set the number_method to (|numberMethod| of parameters) as integer
            set the number_to_choose to (|numberToChoose| of parameters) as integer
            if number_method is 1 then
                set the number_to_choose to my convert_percentage_to_number(number_to_choo
se, count of the input)
            end if
            repeat with i from 1 to the number_to_choose
                set the end of the output_items to some item of the input
            end repeat
        else
            set the output_items to the input
        end if
    end if
```

The debugger lets you step through the script and shows the values of globals, locals, and properties.

You can also insert `log` or `display dialog` statements in the script at points where you want to display current values. If the log statement is inside an application `tell` block, use the `tell me to log` expression instead of the simple `log`. The output of these statements appears in the Console log (not in the Automator log).

For additional debugging information, see the section "Frequently Asked Questions About Debugging Automator Actions" in "Developing an Action" in *Automator Programming Guide*.

Document Revision History

This table describes the changes to *Automator AppleScript Actions Tutorial*.

Date	Notes
2007-07-18	Added debugging information and corrected minor grammatical error.
	In "Building and Testing the Action" (page 45), added information about logging, as well as a link to the section "Frequently Asked Questions About Debugging Automator Actions" in "Developing an Action" in *Automator Programming Guide*.
2005-06-06	New tutorial showing how to create an Automator action using AppleScript.