Chapter 8

ARCHIVING

CHAPTER 8 ARCHIVING

Files and bits coming across a cable are linear. They demand that the data be put in a neat stream of bits. The stream has a beginning and an end. Objects, on the other hand, live in a very complex network of pointers. There is no sense of the beginning or end of a network of pointers.

To put objects in a file or send them across the network to another machine, you need to put then into a linear representation.

Putting objects into a linear representation is known as archiving. Restoring objects from a linear representation is known as unarchiving.

Goal

To understand the archiving mechanism.

Prerequisites

An understanding of how objects can be related to each other in a complex network.

Objectives

At the end of this chapter, you'll be able to:

» Write classes with **initWithCoder:** and **encodeWithCoder:** methods

» Use open and save panels

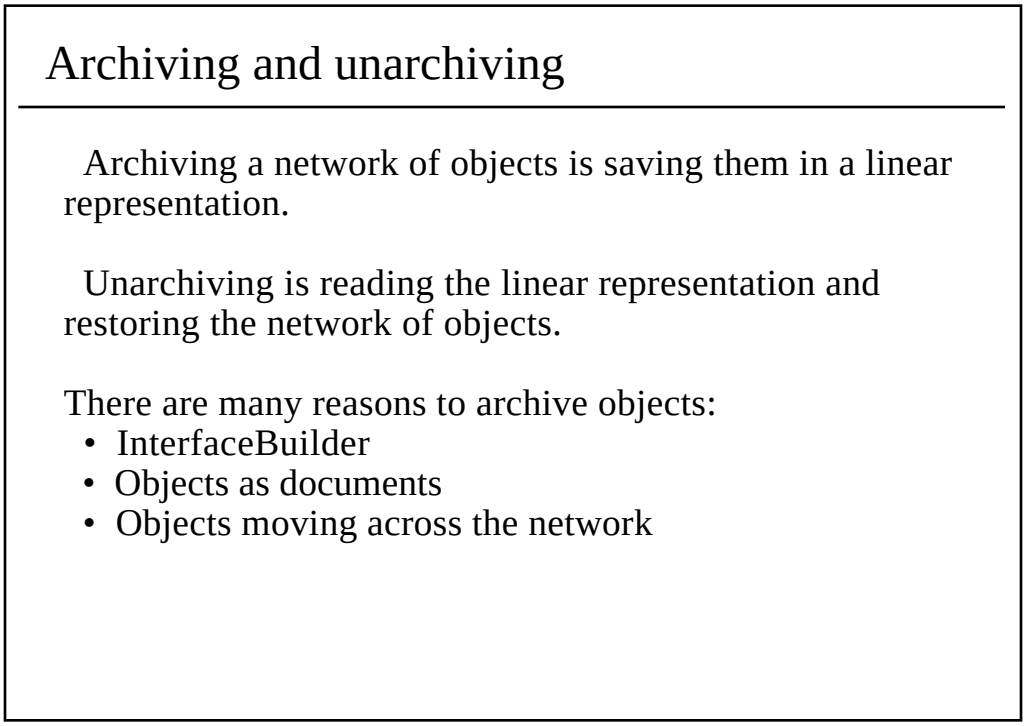
» Save and load files of objects using NSArchiver and NSUnarchiver

Reading

/System/Library/Frameworks/Foundation.framework/Resources/English.lproj/Documentation/Reference/ObjC_classic/...(Protocols-NSCoding)

/System/Library/Frameworks/Foundation.framework/Resources/English.lproj/Documentation/Reference//ObjC_classic/... (Classes-NSCoder, NSArchiver, NSUnarchiver)

/System/Library/Frameworks/AppKit.framework/Resources/English.lproj/Documentation/Reference//ObjC_classic/... (Classes-NSOpenPanel, NSSavePanel)



Archiving and unarchiving

Earlier in the course, you used Interface Builder to create and edit objects. When you were done, you saved them into a nib file. At the time, the nib file was describe as “freeze-dried” objects. Archiving is how they were freeze-dried.

Unarchiving is what happens when the application loads a nib file and brings the objects back to life. This chapter discusses how you can add methods to your objects so they can be archived and unarchived.

Interface Builder is not the only application that uses archiving. When an object is copied from one application to another, perhaps with copy and paste, archiving is typically used.

When objects are moved from one computer to another, archiving is used.

NSCoder

NSCoder is the abstract superclass for all archivers and unarchivers.

NSArchiver and NSUnarchiver are commonly used subclasses of NSCoder.

For an object to be archivable, it must implement two methods:

```
(id)initWithCoder:(NSCoder *)coder;

(void)encodeWithCoder:(NSCoder *)coder;
```

NSCoder

A coder is an object that represents the linear media you are archiving to or unarchiving from. It has methods like (void)**encodeRect:**(NSRect)aRect. An NSRect is a data structure representing a rectangle. When you send a coder the message **encodeRect:**, you are saying, "Here is a rectangle, please put it in the data stream." If the coder represents a file, the data in the rectangle goes into the file.

NSCoder also has the method - (NSRect)**decodeRect:**. Sending this message is like saying "The next thing in the stream of data is a rectangle, please return a new rectangle containing that data."

An object archives itself by encoding its instance variables with a coder. An object unarchives itself by decoding its instance variables from a coder.

If an instance variable points to another object, the object being archived uses NSCoder's **encodeObject:** method. The coder then tells the other object to archive itself, also.

You can imagine how archiving works—tell one object to archive itself. It tells several of its helper objects to archive themselves. They each tell several of their helper objects to archive themselves. Pretty soon an entire network of objects is archived. NSCoder makes sure each object in the network is only archived once.

initWithCoder: and encodeWithCoder:

For your objects to be able to unarchive and archive themselves, they must implement **initWithCoder:** and **encodeWithCoder:**. **initWithCoder:** is used for unarchiving and **encodeWithCoder:** for archiving.

In initWithCoder:, the object asks the coder to decode data on the stream and then sets its instance variables to those values.

```
- (id)initWithCoder:(NSCoder *)coder
{
   [super initWithCoder:coder];
   [self setCompany:[coder decodeObject]];
   [coder decodeValuesOfObjCTypes: "if", &shares, &price];
   return self;
}
```

Again, you're only responsible for instance variables you declare. Call the superclass's **initWithCoder:** to archive its instance variables. The only exception is for direct subclasses of NSObject. Unlike **init**, NSObject has no implementation of **initWithCoder:**.

Notice that this implementation calls an accessor method to set the instance variable **company**. Objects returned from the decodeObject method of a coder are autoreleased. Calling the accessor method to set **company** takes care of retaining the new **company** and releasing the old one. Also notice the **decodeValuesOfObjCTypes** message takes a format string, in a similar manner as a printf statement. In the example "if" stand for integer and float. The codes are similar to those for printf.

In **encodeWithCoder:**, the object tells the coder to encode data from its instance variables.

```
- (void)encodeWithCoder:(NSCoder *)coder
{
   [super encodeWithCoder:coder];
   [coder encodeObject:company];
   [coder encodeValuesOfObjCTypes: "if", &shares, &price];
}
```

Because you're encoding onto a linear stream of bits, you must encode and decode instance variables in the same order. It's a good idea to write **initWithCoder:** and **encodeWithCoder:** at the same time.

Together, i**nitWithCoder:** and **encodeWithCoder:** make up the NSCoding protocol. Instead of declaring these methods, you can simply implement them and adopt the protocol in your class's interface.

@encode

To archive Objective-C types, use encoding codes.

```
[coder encodeValueOfObjCType:"c" at:&myChar];
```

To make the compiler look them up for you, use @encode().

```
[coder encodeValueOfObjCType:@encode(char) at:&myChar];
```

@encode

Part of the goal of using NSCoder is to be able to represent data types in a machine-independent format. This means the coder must know what data type you're asking it to encode or decode.

To support this, each Objective-C type has a code that represents it. These type codes tell the coder what type of data you want it to encode. You can look up the type codes in the documentation, or the Objective-C compiler can look them up for you. Simply use the **@encode()** compiler directive and the compiler takes care of providing the correct type code.Saving and loading with coders.

Saving and Loading with Coders

NSCoder is an abstract class. You never create an instance of NSCoder. Instead, you create instances of its concrete subclasses. Typically, you use NSArchiver for archiving to a file and NSUnarchiver for unarchiving from a file.

Because an entire network of objects can be archived by archiving a single object, to save a network of objects, you simply tell NSArchiver to archive a root object into a file. For example:

```
- (void)saveToFile:(NSString *)filename
{
   [NSArchiver archiveRootObject:myObject
toFile:filename];
}
```

Similarly, an entire network of objects can be unarchived by unarchiving a single root object. So you just tell NSUnarchiver to unarchive a root object from a file.

```
- (void)loadFromFile:(NSString *)filename
{
   [myObject release];
   myObject = [NSUnarchiver
unarchiveObjectWithFile:filename];
   [myObject retain];
}
```

Using save and open panels

Save and open panels are the standard way for users to indicate where they want data written to or read from. You generally don't create instances of save and open panels. Instead, you use a shared instance. Here is some sample code that gets a filename from a save or open panel.

```
NSSavePanel *sp = [NSSavePanel savePanel];
NSString *filename;

if ([sp runModalForDirectory:@"" file:@""]) {
   [self setFilename:[sp filename]];
}

NSOpenPanel *op = [NSOpenPanel openPanel];
NSString *filename;

if ([op runModalForDirectory:@"" file:@""]) {
   [self setFilename:[op filename]];
}
```

See the NSSavePanel and NSOpenPanel class documentation for more details.

Important ideas from this lesson

- To put objects in a file or on a wire, use archiving.
- For an object to be archivable, it must conform to the NSCoding protocol.
- NSCoding declares two methods: **initWithCoder:** and **encodeWithCoder:**.
- Subclasses of classes that conform to NSCoding should call their parent class's implementation of **initWithCoder:** and **encodeWithCoder:** first.
- NSObject does not conform to NSCoding.
- To save a network of objects to a file, you archive a root object.
- Use open and save panels to allow the user to choose the file they want data read from or written to.

REVIEW ARCHIVING

1. Give an example of archiving at work.

2. Class Person, a subclass of NSObject, has two instance variables: **name** is an NSString *, and **points** is an int. In the space below write an **initWithCoder:** and an **encodeWithCoder:** method for Person.

EXERCISE 8.1 ARCHIVER: SAVING DATA

PortfolioManager suffers from one significant flaw. As soon as you quit PortfolioManager, all the data you've entered goes away.

In this exercise you will add support for encoding to the objects in your object model. To save their data when an application is not running, or to transmit data to another application running on a different computer, objects need to be able to encode the data in their instance variables. You'll write **encodeWithCoder:** and **initWithCoder:** methods to support encoding using NSCoder objects supplied with Foundation Framework.

Objectives

After completing this exercise, you'll be able to:

- Write an **encodeWithCoder:** method to encode instance variables
- Write an **initWithCoder:** method to decode instance variables and properly initialize a new object with those values
- Use NSOpenPanel to choose a file to unarchive
- Use NSSavePanel to choose a file for archiving

Exercise

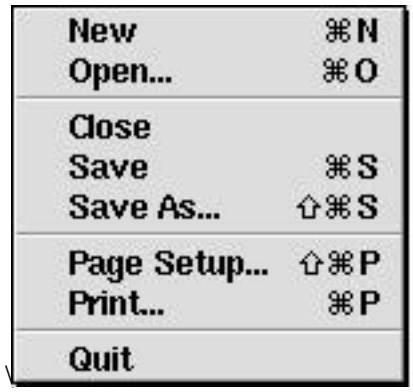
1. In order to support archiving to a file, an application needs to implement two action methods: **open:** and **save:**. Add the following method declarations to **PortfolioController.h**:

```
(void)open:(id)sender;
(void)save:(id)sender;
```

2. Open the PortfolioManager.nib file of the Portfolio Manager project.

3. Drag the **PortfolioController.h** icon from ProjectBuilder onto InterfaceBuilder's window. InterfaceBuilder will parse the file, adding the new action methods.

4. Drag a File menu from the Interface Builder's Menus palette onto PortfolioMabnager's main menu.



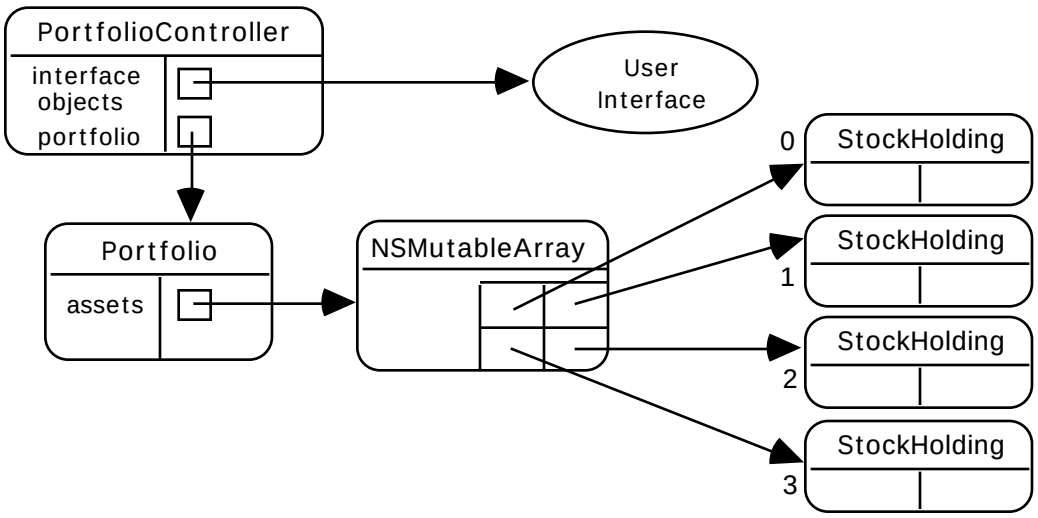
Interface Builder's Menus palette contains a number of standard submenus. One of these is the File menu. It provides a number of menu items often used for managing files. By using the supplied File menu, you ensure that your application has a consistent look and feel with other applications. By default, most menu items are disabled.

5. Connect the Open menu item to the instance of PortfolioController. Set the action to **open:**.

6. Connect the Save menu item to the instance of PortfolioController. Set the action to **save:**.

The user interface for archiving is now complete! There's no need to create a custom open or save panel since these are provided as part of the Application Kit. The remainder of the exercise consists of teaching your model classes how to archive themselves, and implementing the **open:** and **save:** methods you've declared in PortfolioController.

7. NSArchiver can archive a root object and all its objects to a file using the **archiveRootObject:toFile:** method. Below is a diagram of Portfolio Manager's objects. Identify the root object, as well as the classes you need to make conform to the NSCoding protocol.



8. Add **encodeWithCoder:** methods to the classes you've identified. Note that the Asset class already implements an **encodeWithCoder:** method -- make sure any subclasses of Asset call the superclass's implementation first.

9. Write **initWithCoder:** methods for the model classes you've identified. These methods should mirror the **encodeWithCoder:** methods. Again, if the class's superclass implements **initWithCoder:** be sure to call the superclass's implementation first.

10. Now that the model objects know how to encode and decode themselves, it's time to add support for writing those objects to a file. Implement the **save:** method in PortfolioController. **save:** needs to get a file name from the user using NSSavePanel, and save the root object to the file using NSArchiver's **archiveRootObject:toFile:** method. Check the diagram from Step 7 if you're not sure which is the root object.

11. Implement the **open:** method in PortfolioController. **open:** must also do two things: get a file name from NSOpenPanel, and set the root object to the object it gets from NSUnarchiver. Use NSUnarchiver's **unarchiveObjectWithFile:** method to unarchive the root object. Use PortfolioController's **setPortfolio:** accessor method to set the newly unarchived object.

    Remember that objects returned from methods other than **alloc** or **copy** should be considered autoreleased. That's why it's important to use the **setPortfolio:** method. **setPortfolio:** takes care of getting rid of the old portfolio, retaining the new portfolio object, and coordinating the user interface.

12. The open method also has to update the user interface to display the new data. Call PortfolioController's **fillFields** method to display the selected stock holding from the newly loaded portfolio.

13. Build and test your application. You should be able to save a portfolio to a file and later read in the saved information.

Enhancements

» Implement a **new:** method that clears the current portfolio and user interface and allows the user to start over with a new portfolio

» Add support for Save As. Factor out common code from **save:** and **saveAs:** into a **saveToFile:** method declared like this:

```
(void)saveToFile:(NSString *)filename;
```

» Add support for Revert to Saved. Factor out common code from **open:** and **revertToSaved:** into an **readFromFile:** method declared like this:

```
(void)readFromFile:(NSString *)filename;
```