Chapter 7

FOUNDATION

CHAPTER 7 FOUNDATION

The Foundation Framework defines a base layer of Objective-C classes. In addition to providing a set of useful primitive object classes, it introduces several paradigms that define functionality not covered by the Objective-C language.

—Foundation, Introduction

Goal

To be able to effectively use fundamental classes and concepts from the Foundation Framework.

Prerequisites

Working knowledge of objects and pointers.

Objectives

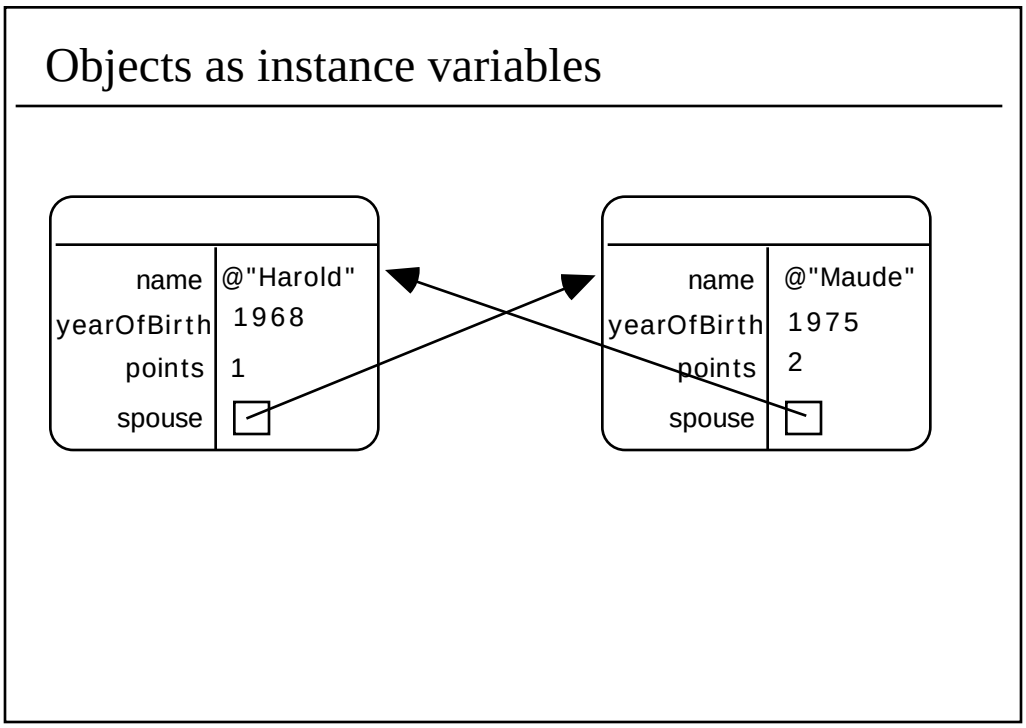
At the end of this chapter, you'll be able to:

- Distinguish between simple attributes and relationships
- Use the NSArray and NSMutableArray collection classes
- Properly use autorelease when returning objects from methods

Reading

The following reference contains more information about the Foundation Framework:

/System/Library/Frameworks/Foundation.framework/Resources/English.lproj/Documentation/Reference/ObjC_classic/IntroFoundation.html



Objects as instance variables

Objects can serve two different roles when used as an instance variable of another object. Objects that store values, like NSString and NSCalendarDate, are used for instance variables that represent simple attributes of the object. A simple attribute is an attribute that resolves to a value—for example a name, number, or date.

Instance variables that point to business objects generally represent relationships. A relationship is a pointer to a related object. The related object doesn't store data for you, it's simply a related object.

In the example of a Person object, **name** is a simple attribute. Even though the instance variable points to an object, an instance of NSString, conceptually **name** resolves to a value. **spouse** is a relationship. It points to another Person object.

The Foundation Framework introduces a number of classes that are very useful for storing simple attributes. NSString and NSCalendarDate are two that you will use frequently.

Accessor methods revisited

The accessor methods for instance variables that represent simple attributes and those that represent relationships are different. Because simple attributes are part of an object, it's necessary to get a clean copy of the objects used to store simple attributes when you set the instance variable. If someone calls **setName:** on a Person and passes in a string, you want to be sure that string will never change. For this reason, accessor methods for simple attributes should copy the passed in object. For example:

```
(void)setName:(NSString *)aName
{
   if (name != aName) {
      [name release];
      name = [aName copy];
   }
}
```

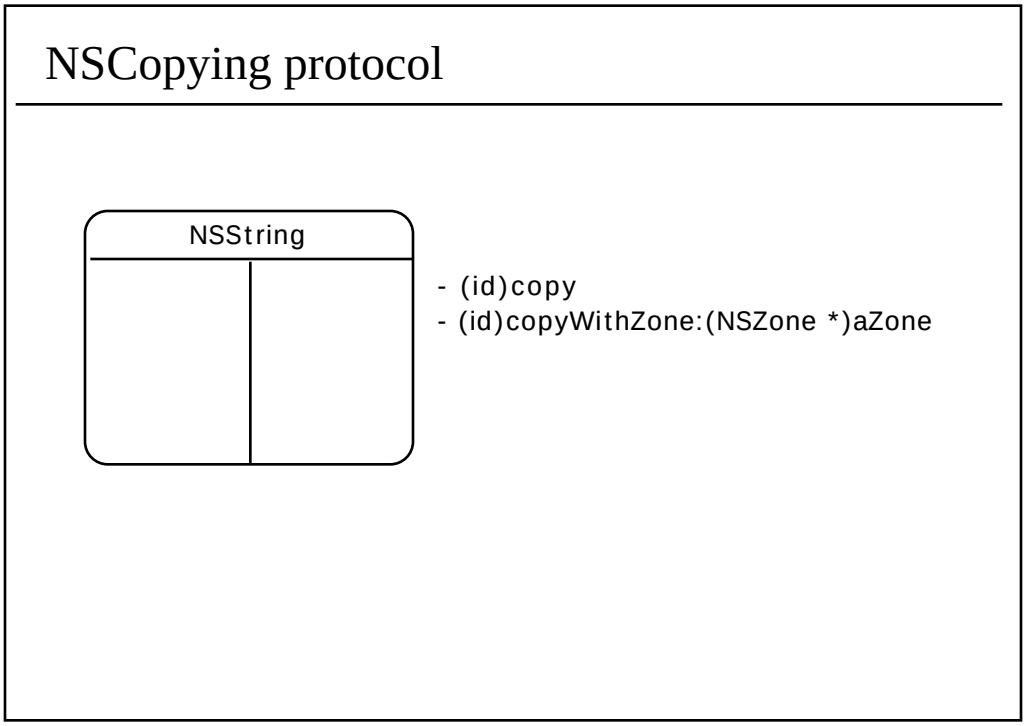
When creating a relationship to another object, you don't want to copy the related object. Imagine two Person objects, **personA** and **personB**. You wish to create a relationship between them, indicating that **personB** is **personA's** spouse:

```
[personA setSpouse:personB];
```

When you send **personA** the **setSpouse:** method, you expect **personA** to store a reference to **personB**, but not to create a new, private copy of **personB**. Therefore, Person's **setSpouse:** method should look like this:

```
(void)setSpouse:(Person *)aPerson
{
   if (spouse != aPerson) {
      [spouse release];
      spouse = [aPerson retain];
   }
}
```

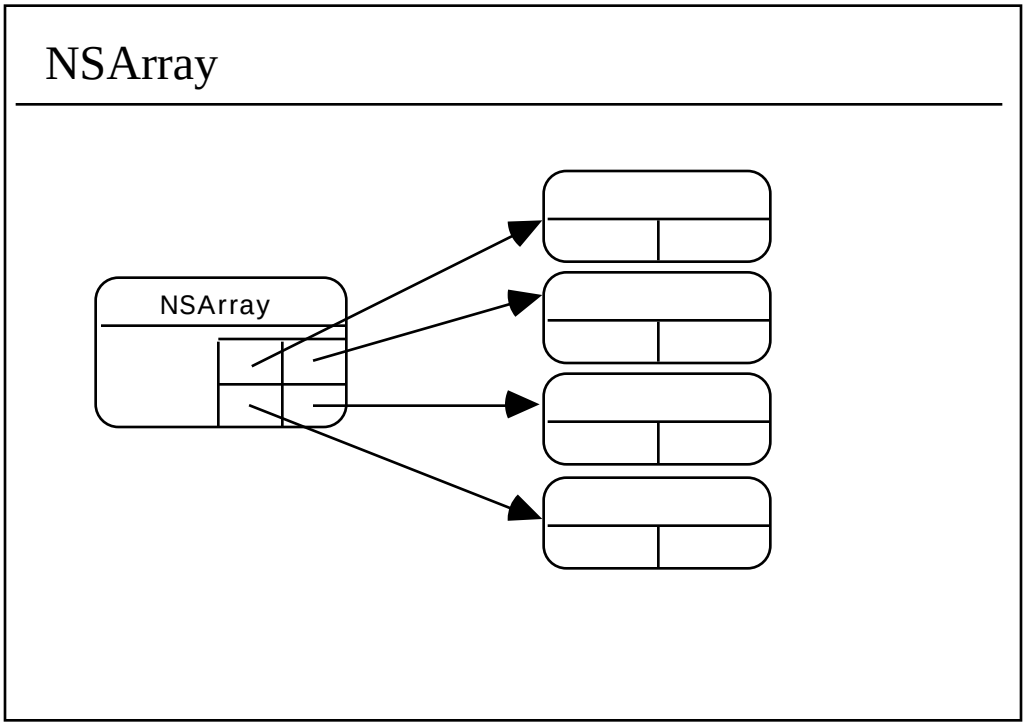
Accessor methods for instance variables that embody relationships should keep a reference to the passed in object, but not create a copy.



NSCopying protocol

To make it easier to copy them, all classes in the Foundation Framework that are used to store values conform to the NSCopying protocol. The NSCopying protocol says that any object that conforms to it must implement the method **copyWithZone:**. As a convenience, the NSObject class defines the method copy which simply invokes copyWithZone: with a default zone. When you send an object a **copy** message, you get back a retained object whose data is an exact copy of the original object's data.

Other objects may implement the NSCopying protocol. However, for objects with data that is more complex than a single value, it can be difficult to know what to do when copying them. For example, should the Person object return a copy of itself with a copy of its spouse, or just another reference to its spouse? Always check the documentation before using copy on an object that stores complex information.

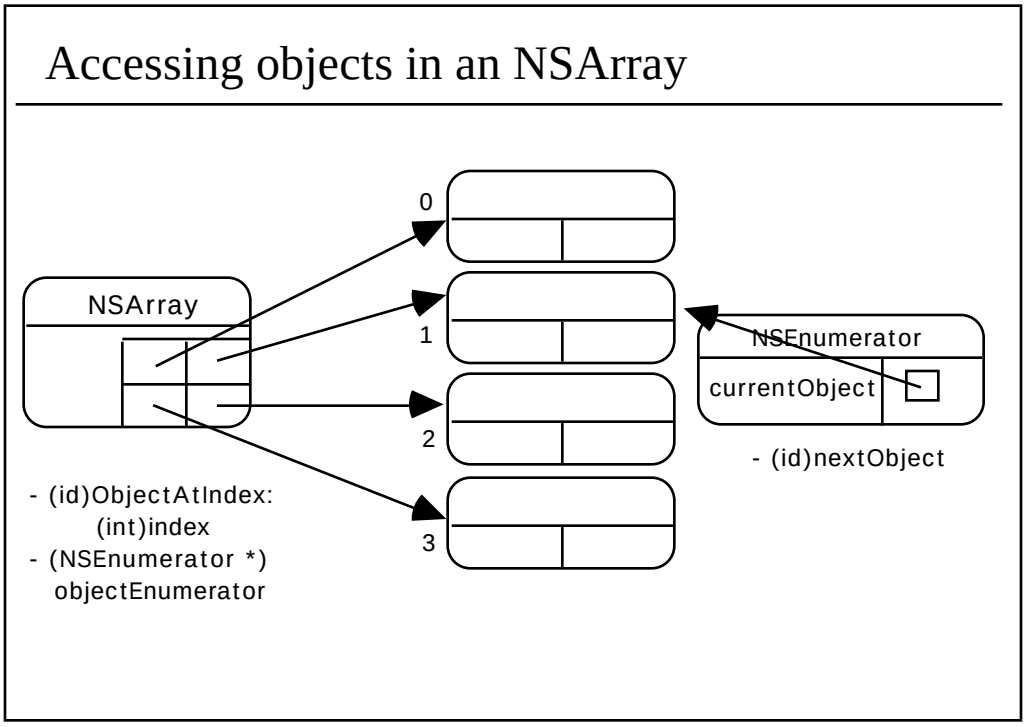


NSArray

In many situations, it's desirable to keep an ordered list of objects. The Foundation Framework provides the NSArray class to keep such a list. When an object needs to return an ordered collection of objects, it generally uses an NSArray. NSArray is a collection class.

Normally, instances of NSArray cannot be modified. This is done to preserve encapsulation. When an NSApplication returns its list of NSWindows, the NSApplication is exposing some of its instance variables. If you could change the windows in an application by modifying the array, it would break encapsulation. Therefore, NSArray is not modifiable. Foundation refers to objects that can not be changed as **immutable** objects.

Passing around immutable objects is easier than passing around mutable objects, because they're guaranteed to never change. This can eliminate excess copying.



Accessing Objects in an NSArray

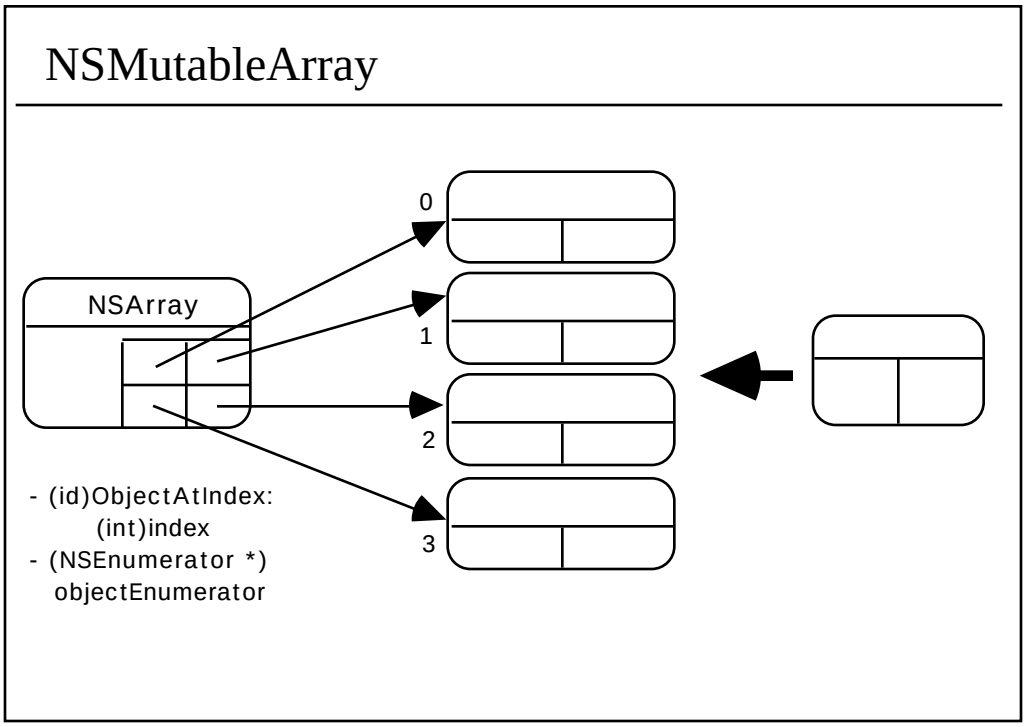
Objects in an NSArray are ordered by index. To access the objects in an NSArray, you can simply ask the array for the object at a given index. For example:

```
id firstObject;
NSArray *theArray;

firstObject = [theArray objectAtIndex:0];
```

Index numbers start at 0.

If you need to traverse the contents of the entire array, you could simply create a for loop to do so. However, NSArray provides an object to do the job for you—an instance of NSEnumerator. You use NSEnumerator like this:

```
NSEnumerator *theEnumerator;
id theObject;

theEnumerator = [theArray objectEnumerator];
while (theObject = [theEnumerator nextObject]) {
   // do operations using theObject here
}
```

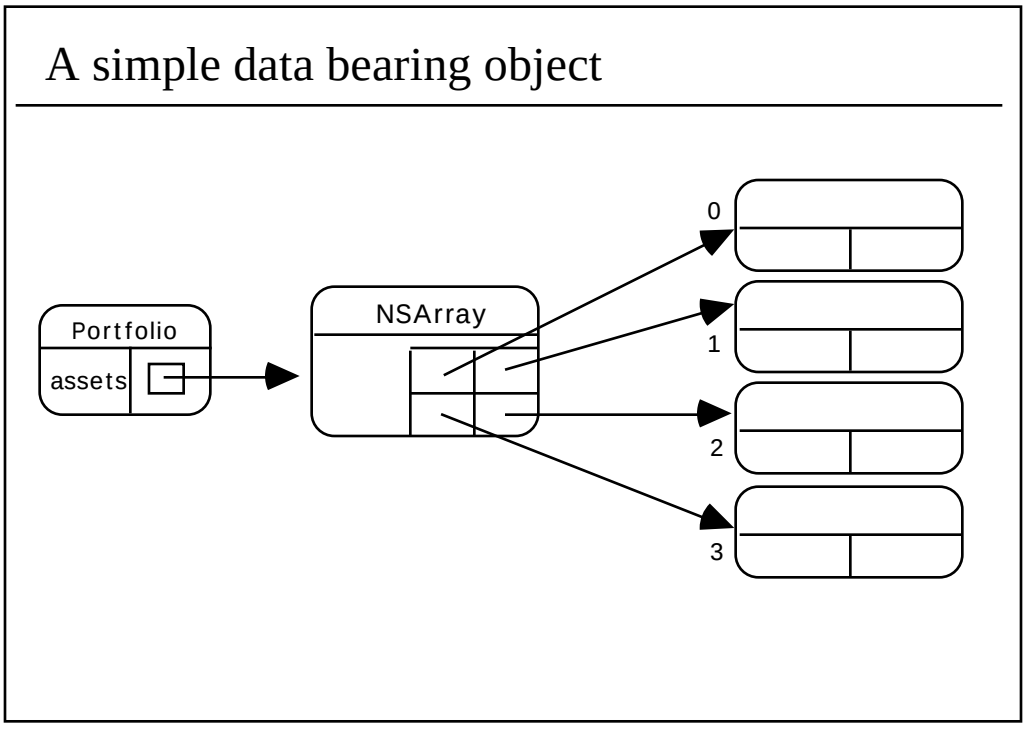


NSMutableArray

Sometimes it's desirable to modify an array incrementally, instead of creating an immutable array. The Foundation Framework provides NSMutableArray to meet this need. NSMutableArray provides all the methods of NSArray, plus it adds method for adding and removing objects from the array. Objects can be inserted at a particular index, or simply added to the end of the array. NSMutableArray dynamically allocates storage to hold references to as many objects as you add to it.

When you add an object to an NSMutableArray, the array sends your object a retain message. This is because objects in an array are retained by the array. This is true regardless of whether the array is a mutable or immutable array. When you remove an object from an array, the array sends it a **release** message. An array also releases its objects when the array itself is deallocated.

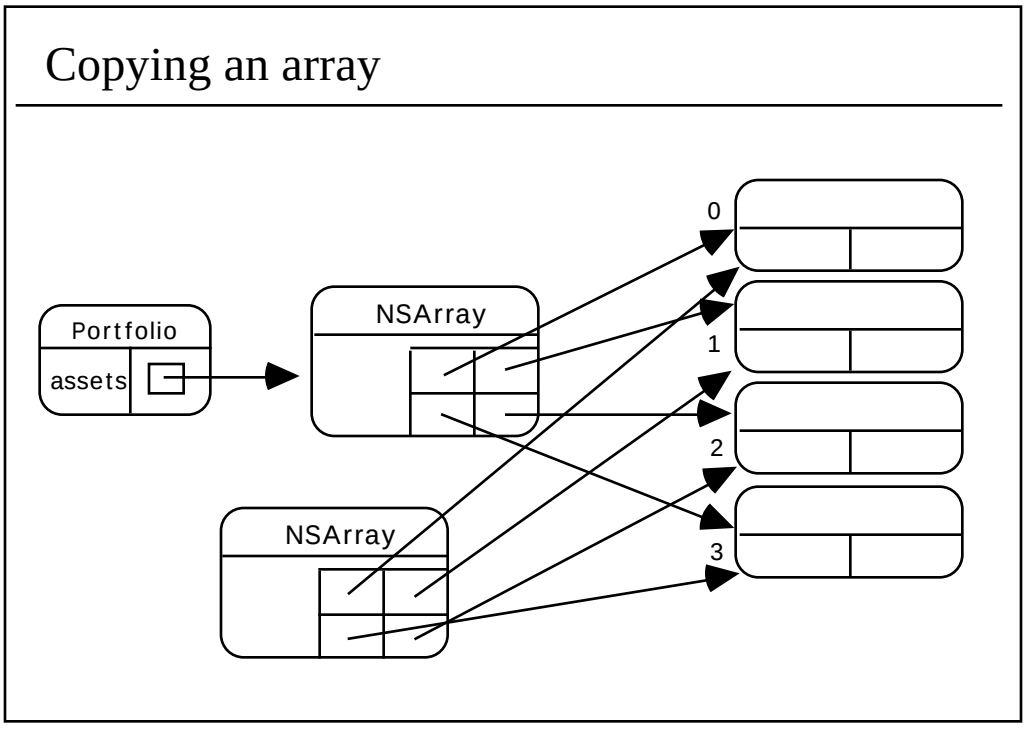
NSMutableArray allows you to get an NSEnumerator to enumerate the array's objects. Because NSEnumerator peeks into the innards of an array for efficiency reasons, you should never modify an array while simultaneously using an NSEnumerator to step through its member objects. Not following this rule can lead to unpredictable results.



A simple data bearing object

NSMutableArray provides an easy way to implement a simple data bearing object. Create an object that uses an instance of NSMutableArray to manage its data. You add value by providing additional methods that perform operations on the objects in the array.

In some cases, it's desirable to give a client a list of all the objects in the NSMutableArray. For example, a Portfolio object might manage a list of Assets. A client might want to take this list of Assets and add them to another Portfolio object, or perform some other operation on them. The question is, what should the Portfolio return? If it returns the NSMutableArray itself, it's broken encapsulation. If it sends the NSMutableArray a **copy** message, not only is the array itself copied, but so are all the objects in the array.



Copying an array

NSMutableArray and NSArray make assumptions about what you want to do when you send them a copy message. Specifically, NSMutableArray assumes that you want not only a copy of the pointers to the objects, but also a copy of the objects themselves. So NSMutableArray performs a deep copy. NSArray assumes you just want to retain the array one more time, so it simply increments its retain count.

In the example of Portfolio returning an array of the objects in its NSMutableArray, the right thing to do is to copy the array itself—the pointers to the objects. This is cleanly accomplished using NSArray's i**nitWithArray:** method. Portfolio's **assets** method might look like this:

```
- (NSArray *)assets
{
   NSArray *returnedArray;
   returnedArray = [[NSArray alloc]
initWithArray:assets];
   [returnedArray autorelease];
   return returnedArray;
}
```

Autorelease revisited

Portfolio's **assets** method created a completely new object that it then used as its return value. As far as Portfolio is concerned, **returnedArray** is simply a return value and should go away. Portfolio claims no ownership of the object, even though it created it. Following the rules from Chapter 4, Portfolio is responsible for releasing **returnedArray**, because Portfolio created it using **alloc**.

If Portfolio simply sends **returnedArray** a **release** message, **returnedArray** will deallocate itself immediately. This would make it rather useless as a return value. However, once Portfolio returns **returnedArray**, there's no place in the code for Portfolio to release it. The method call is over.

To get around the problem, Portfolio uses **autorelease**. This way, Portfolio has done its job—**returnedArray** has been marked for release. However, it's been marked for **future** release, so it's still useful as a return value.

Here is the rule to follow when creating a new object to use as a return value:

Send **autorelease** to any object you create using **alloc** or **copy** for use as a return value.

Protocols

A **protocol** is a set of methods that can be implemented by any class. For example, the NSCopying protocol defines the method that any class needs to implement to support copying. It's possible to test whether a particular object conforms to a given protocol, and to do type checking based on adopted protocols. This allows many objects throughout the class hierarchy to respond to the same set of messages in a formal way, without having to inherit them all from the same superclass.

Protocols are defined using a syntax very similar to the method declarations in a header file. For example:

```
@protocol NSCopying

- (id)copyWithZone:(NSZone *)aZone;

@end
```

Classes can adopt protocols using the following syntax:

```
@interface Portfolio:NSObject <NSCopying>
```

Multiple protocols are separated using commas.

Type checking for protocols is done using a similar syntax. The following variable declaration declares a variable that refers to any class that adopts the NSCopying protocol:

```
id <NSCopying> copyableObject;
```

Foundation defines a number of protocols, listed in the documentation. Other frameworks may also define protocols. Check the documentation for each framework to see what protocols it defines.

Important ideas from this section

- Instance variables that point to objects can represent simple attributes or relationships.
- Simple attributes resolve to a value. NSString and NSCalendarDate are two classes typically used to hold simple attributes.
- Relationships are pointers to other business objects.
- Accessor methods that set simple attributes should copy the passed-in object.
- Accessor methods that set relationships should retain the object.
- NSArray stores an ordered list of objects.
- Objects in an NSArray can be accessed by index.
- Objects in an NSArray can be enumerated using NSEnumerator.
- NSMutableArray is an array that can be modified.
- Arrays retain the objects they contain.
- To copy the pointers of an array, use **alloc** and **initWithArray:**.
- Protocols provide a formal way for many classes to implement the same methods.

REVIEW FOUNDATION

1. If Asset has a name instance variable that points to an NSString, is **name** a simple attribute or a relationship?

2. Write an implementation for Asset's **setName:** accessor method.

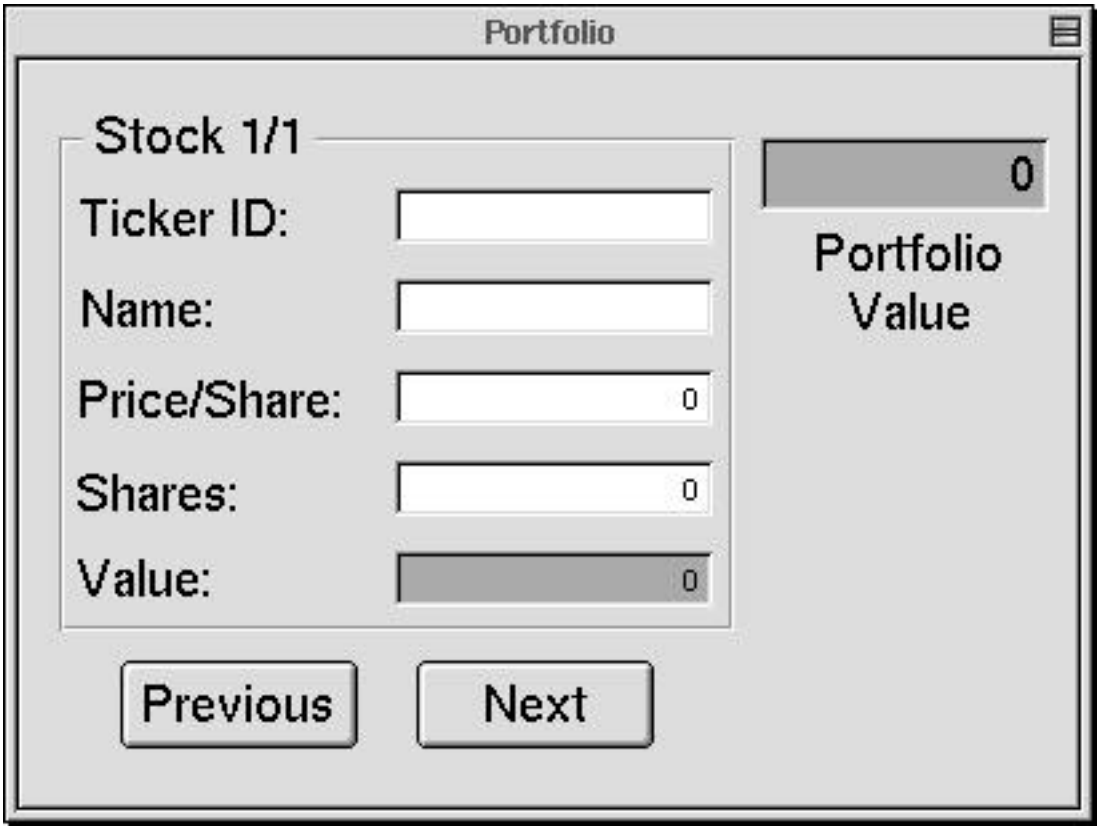
3. Given that **assetArray** is an instance of NSArray, write a message call that returns the object in **assetArray** at index 3.

4. Write a code fragment that sends the **value** message to each object in **assetArray:**.

5. What is the difference between NSArray and NSMutableArray?

6. What are protocols used for?

EXERCISE 7.1 IMPLEMENTING PORTFOLIO



For the previous exercises, the Portfolio class was provided to you. In this exercise, you get a chance to use what you've learned about the Foundation Framework, and specifically NSArray, to implement Portfolio yourself.

Objectives

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

- Use an instance of NSMutableArray to manage data storage of objects
- Use an NSEnumerator to iterate over the contents of an NSArray
- Copy the pointers in an array without copying the contents of the array

Exercise

1. Remove **Portfolio.h** and **Portfolio.m** from your PortfolioManager project:

1. Make a new **Portfolio.h** file in your PortfolioManager project:

```
#import <Foundation/NSObject.h>

#import “Asset.h”
#import <Foundation/Foundation.h>

@interface Portfolio : NSObject
{
   NSMutableArray *assets;
}

- (id)init;
- (void)dealloc;

- (void)addAsset:(Asset *)anAsset;
- (void)removeAssetAtIndex:(int)index;
- (Asset *)assetAtIndex:(int)index;
- (int)numberOfAssets;
- (NSArray *)assets;
- (double)value;

@end
```

   Portfolio uses an NSMutableArray to store a number of Assets. Access to this array is granted using methods with names very similar to those of NSMutableArray itself. Don’t worry about adopting the NSCoding protocol for this exercise.

2. Create a new **Portfolio.m** file in your project.

3. Implement the **init** and **dealloc** methods. You need to create an NSMutableArray in **init** and release it in **dealloc**.

4. Implement methods for accessing the array. These methods all have functional equivalents in NSArray or NSMutableArray. Your methods should simply pass the arguments along.

5. Write an **assets** method. **assets** returns a list of all Assets currently managed by the Portfolio. Make sure you return an NSArray, not the NSMutableArray. Be sure to properly autorelease any objects you create for use as a return value.

6. Write a **value** method that computes the Portfolio’s value by adding up the value of each Asset in the Portfolio. Your **value** method should iterate over each Asset, calling each ones **value** method in turn and adding up the return values.

7. Build your project and run PortfolioManager. Check to make sure everything still works as expected.