Chapter 6

INHERITANCE

CHAPTER 6 INHERITANCE

The easiest way to explain something new is to start with something old. If you want to describe what a "schooner" is, it helps if your listeners already know what "sailboat" means. If you want to explain how a harpsichord works, it's best if you can assume your audience has already looked inside a piano, or has seen a guitar played, or at least is familiar with the idea of a "musical instrument."

With this in mind, object-oriented programming languages, permit you to base a new class definition on a class already defined. The base class is called a superclass; the new class is its subclass. The subclass definition specifies only how it differs from the superclass; everything else is taken to be the same.

—Object-Oriented Programming and the Objective-C Language

When behavior is inherited from another class, the code that provides that behavior does not have to be rewritten. This may seem obvious, but the implications are important. Many programmers spend a large percentage of their time rewriting code they have written before—for example, to search for a pattern in a string or to insert a new element into a table. Using object-oriented techniques, these functions can be written once and reused.

—Timothy Budd, An Introduction to Object-Oriented Programming

Goal

To explore the issues involved in writing a subclass.

Prerequisites

The ability to create a class.

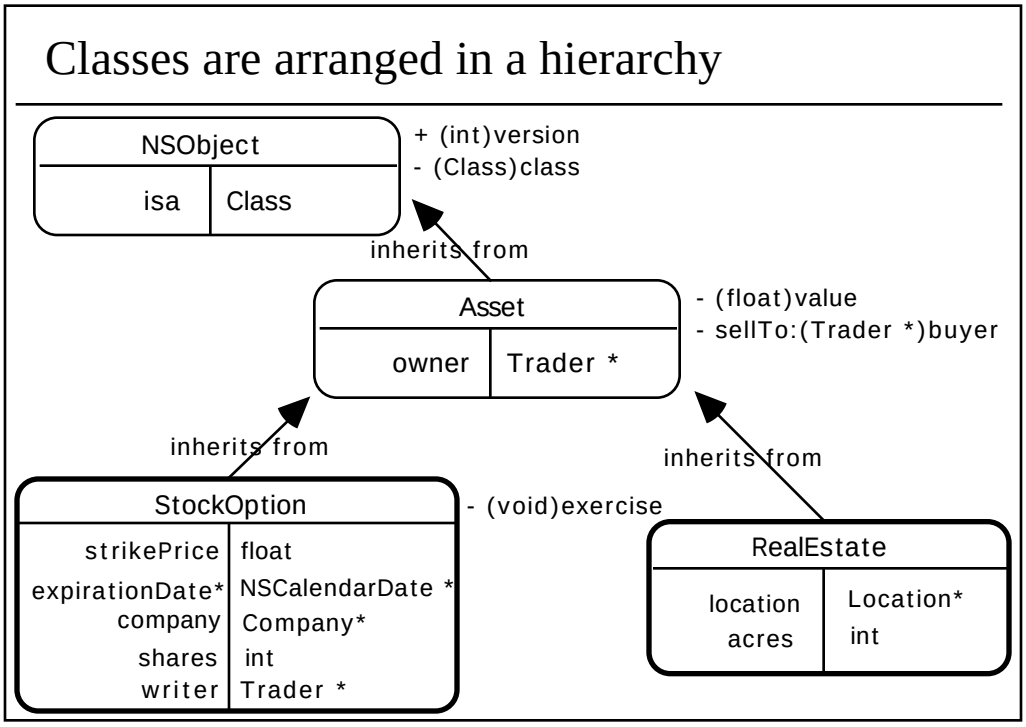
Objectives

At the end of this Chapter, you'll be able to write a subclass that properly initializes and frees its instance variables.

Reading

You can find more information about inheritance in:

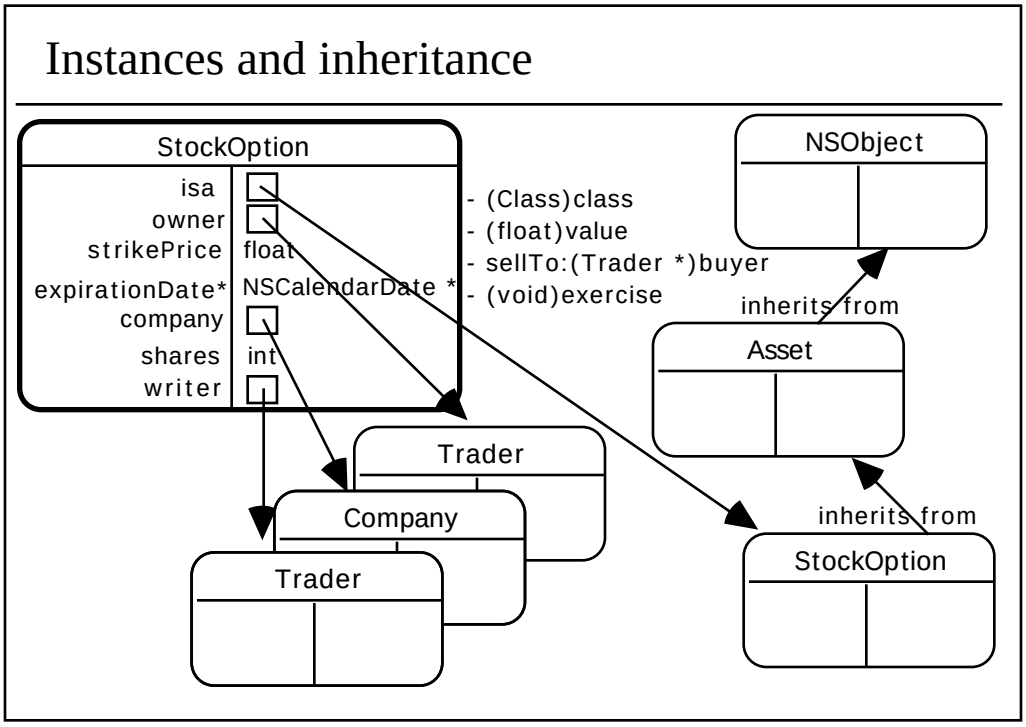
/System/Documentation/Developer/TasksAndConcepts/ ObjectiveC



Classes are arranged in a hierarchy

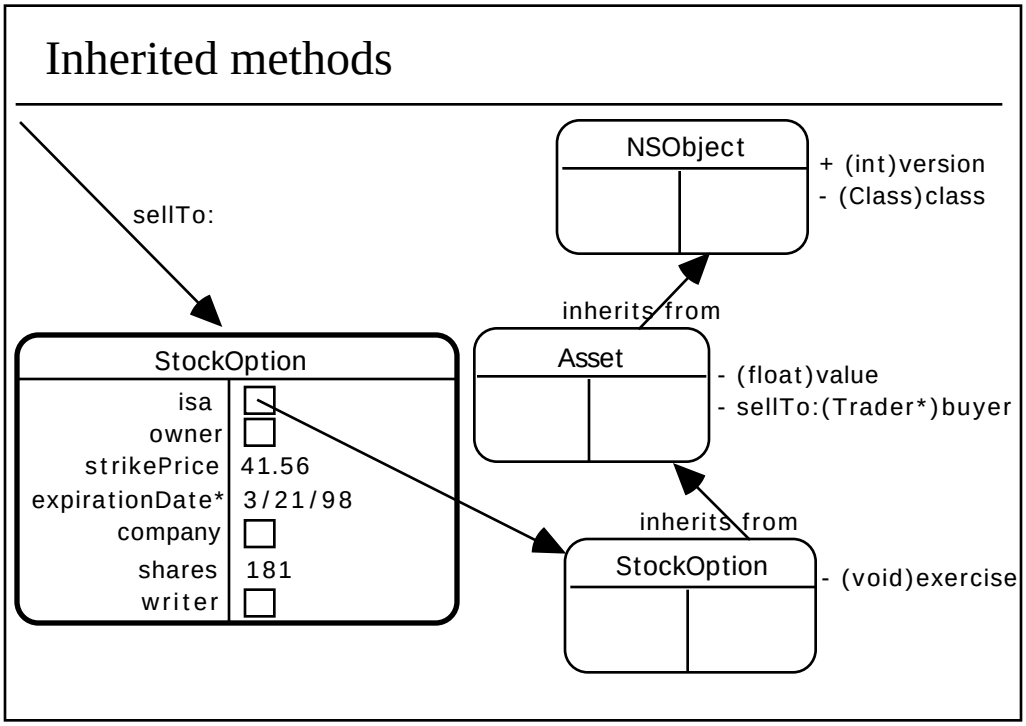
Each class, except NSObject, has exactly one superclass. Of course, a class can have several subclasses. Each class inherits all the methods and instance variables of its superclass.

In this hierarchy, NSObject is at the top and the most specialized classes tend to be at the bottom. In a well-designed hierarchy, if class A is a subclass of class B, you should feel comfortable saying “A is a type of B.” For example, realestate is a type of asset. Thus the class RealEstate might be a good subclass of Asset.



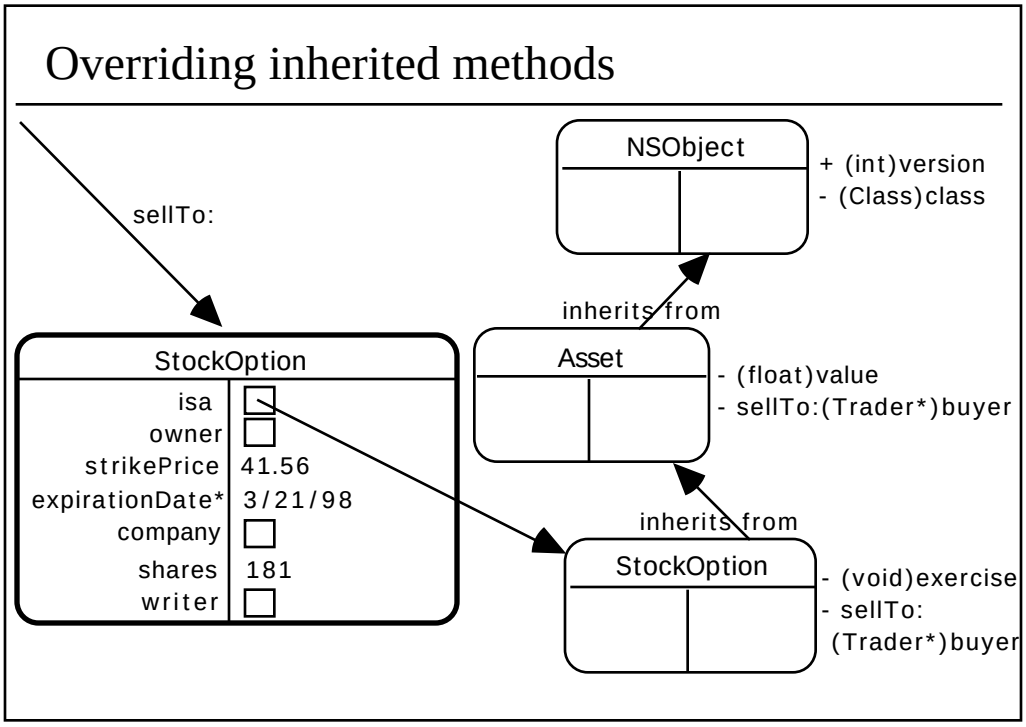
Instances and inheritance

An instance of class A has all the instance variables and methods introduced in class A, and all those introduced in class A's superclass, and those introduced in class A's superclass's superclass, and so on.



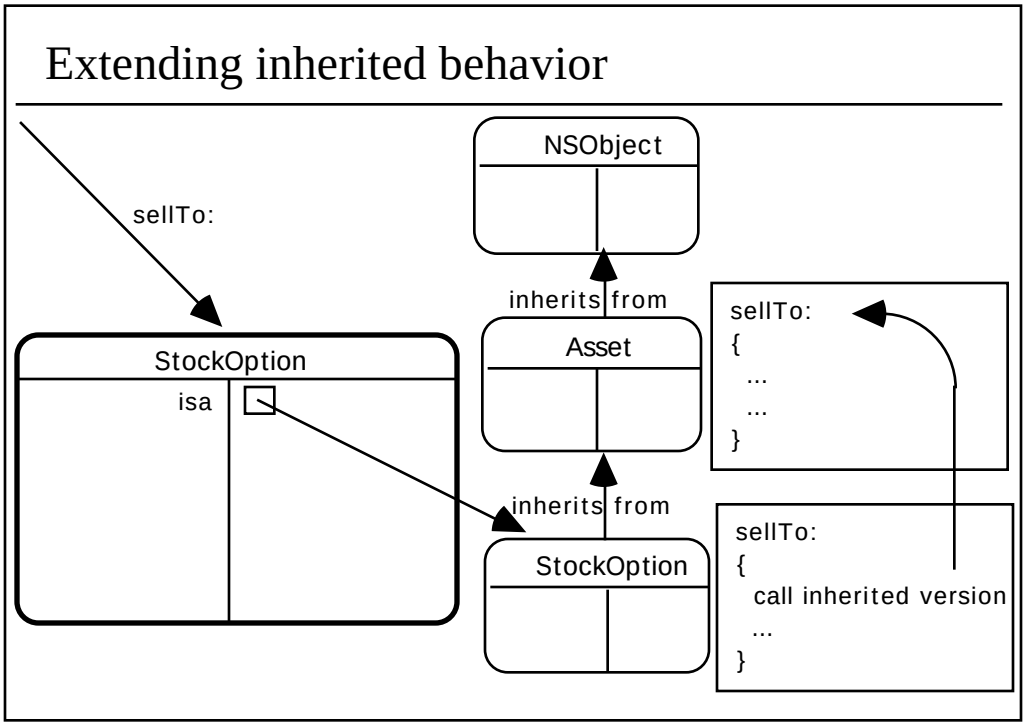
Inherited methods

When an object receives a message, it checks to see if its class implements that method. If not, it checks with its superclass. If it gets all the way to the top of the hierarchy without finding an implementation of the method, it issues an exception.



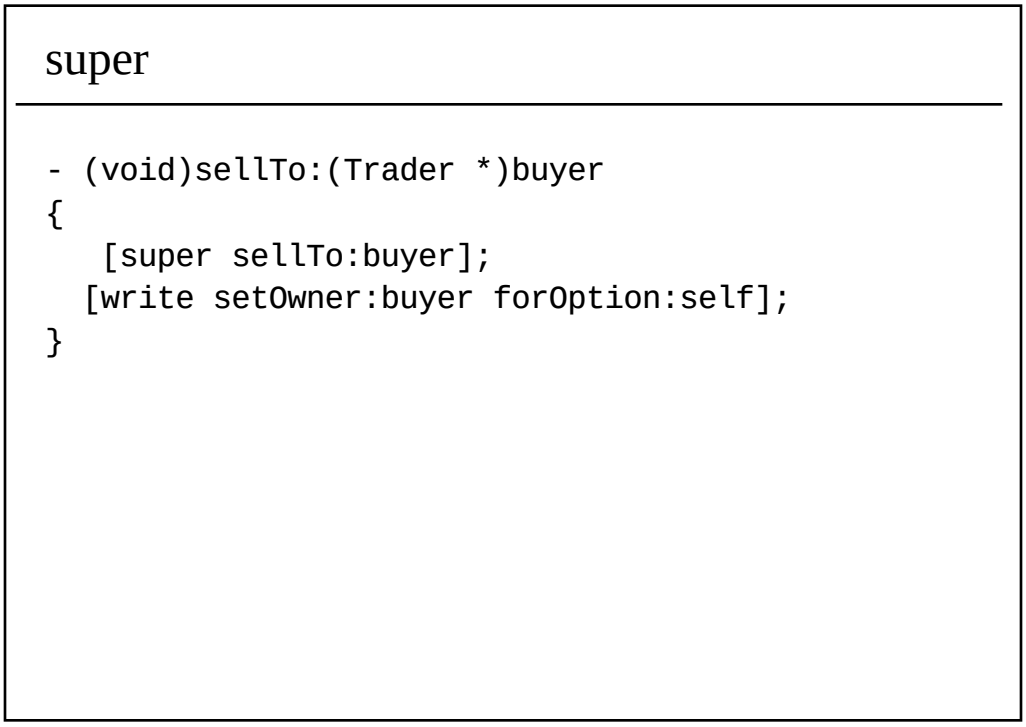
Overriding inherited methods

Because objects search for method implementations from the bottom of the hierarchy to the top, classes lower in the hierarchy can override the implementation of the superclasses. As an instance searches up the hierarchy, it finds the lower implementation first. The implementation higher up in the hierarchy is not executed.



Extending inherited behavior

It's fairly common to create a subclass and find that the superclass's version of a method is good for your subclass, except for some extra work you need to do in the subclass. What you want to do is use the implementation of the superclass and extend it, instead of completely rewiring the method.

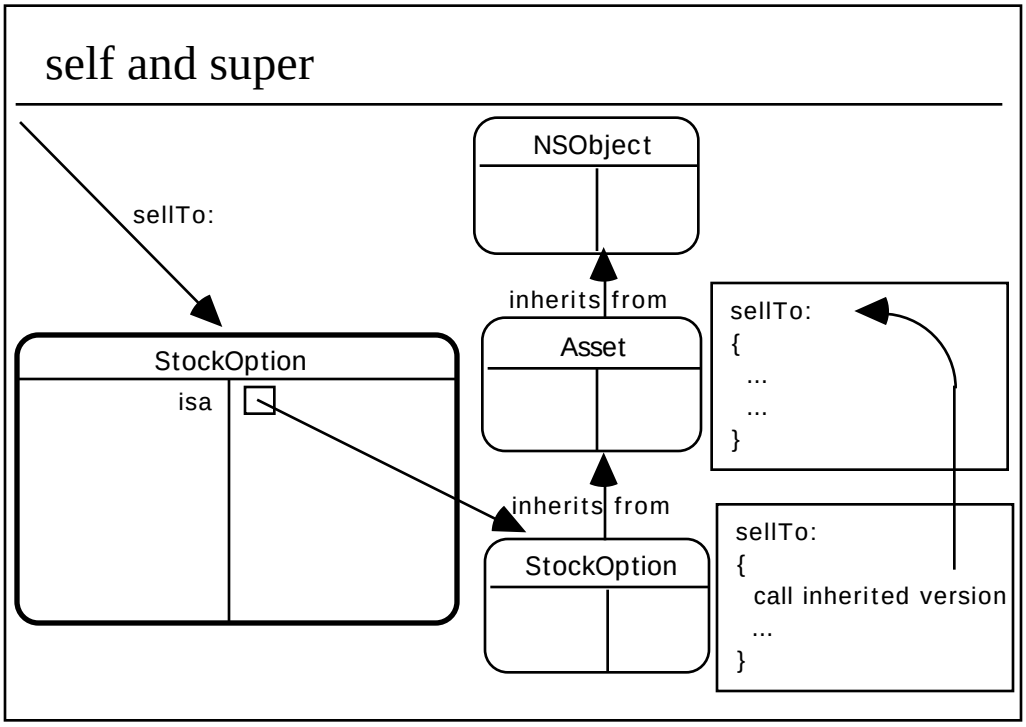


super

Using **super**, a class can call the inherited version of a method. For example:

```
- (void)sellTo:(Trader *)buyer
{
   [super sellTo:buyer];
   [write setOwner:buyer forOption:self];
}
```

A message sent to **super** starts searching for an implementation with the object's superclass, instead of the object's class.



self and super

It's important to understand the difference between **self** and **super**. **self** is a variable that points to a distinct object—the object currently executing a method. **super** is a way of calling the implementation of a method from your superclass. **super** is relative to what class has the implementation. **self** always points to a concrete object.

For example, take StockOption and Asset. If an instance of StockOption receives a **sellTo:** message, it begins executing the implementation it finds first in the class hierarchy—in this case, the implementation defined in StockOption. At this point, **self** points to the instance of StockOption, and **super** refers to the superclass of StockOption, Asset.

Now imagine StockOption's **sellTo:** method called Asset's implementation of **sellTo:** using **super**. At this point, Asset's **sellTo:** method is executing. **super** now refers to Asset's superclass, NSObject. However, **self** still points to the instance of StockOption. Again, **self** is a real variable that points to a specific object. **super** is a way of calling the implementation of a method in your superclass.

alloc and init

When a new instance of any class is created all instance variables (except isa) are set to zero:

- Numbers are 0.0
- Characters are '\0'
- Pointers are nil

The init method is responsible for initializing these to a usable value.

alloc and init

The class method alloc allocates memory for objects. It uses the information in the class to calculate how many bytes of memory to allocate for instances of that class. It then sets all the instance variables to zero.

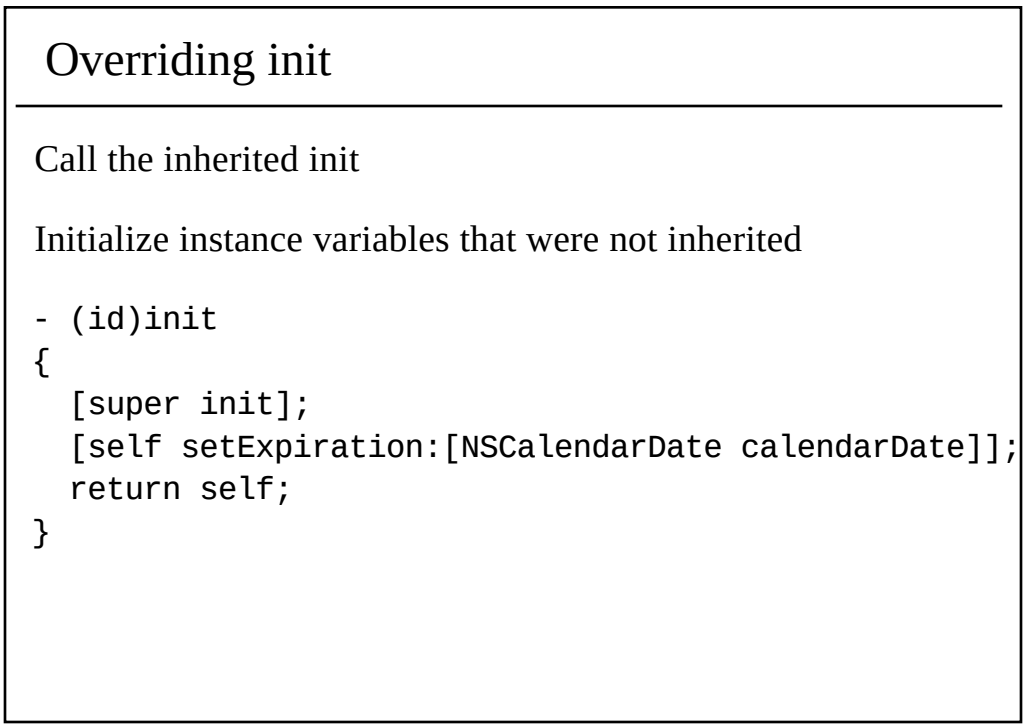
For most classes, it's inappropriate for instance variables to start out as zero. If an instance variable used for someone's name is declared as an NSString, the variable will initially be a nil pointer. A more appropriate value would be an instance of NSString whose contents were "". Having a nil pointer as an instance variable could easily lead to strange bugs.

To make sure all objects are initialized with valid values for their instance variables, an initialization method is typically called right after **alloc**. NSObject declares and implements a method called init that takes care of initialization.

A typical situation where **init** is useful is when an object uses another object as an instance variable. For example, a receipt object might need an array to hold references to all the items purchased. Thus, for the receipt object to be usable, it must allocate a new array and set an instance variable to point to that instance.

OpenStep requires you to initialize an object immediately after allocating it. That is why code for object creation generally looks like this:

```
myReceipt = [[Receipt alloc] init];
```



Overriding init

Since classes are arranged in a hierarchy, when you create a class it may inherit instance variables from several other classes. While you could carefully study the **init** method of each super class and write an **init** method that initializes all the instance variables, it is much easier to simply call the inherited **init** method.

So, the first step in any **init** method is to call the initializer of your super class:

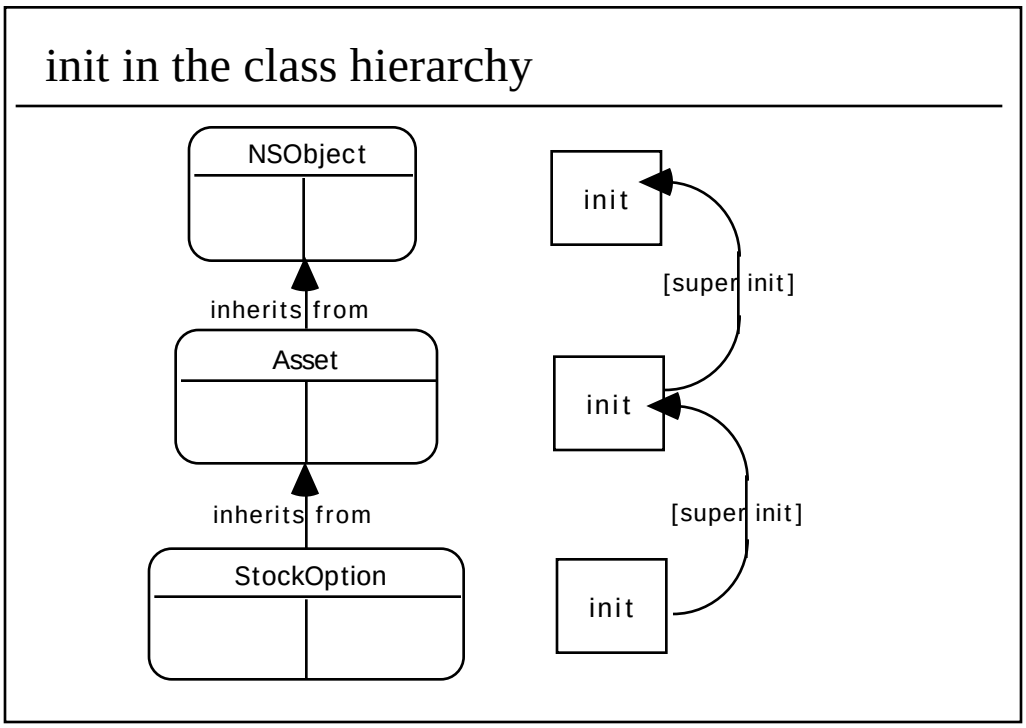
```
[super init];
```

Once you have called the superclass's initializer, you can do further initialization. For example, setting the expiration to today's date:

```
[self setExpiration: [NSCalendarDate calendarDate]]
```

Finally, return the initialized instance:

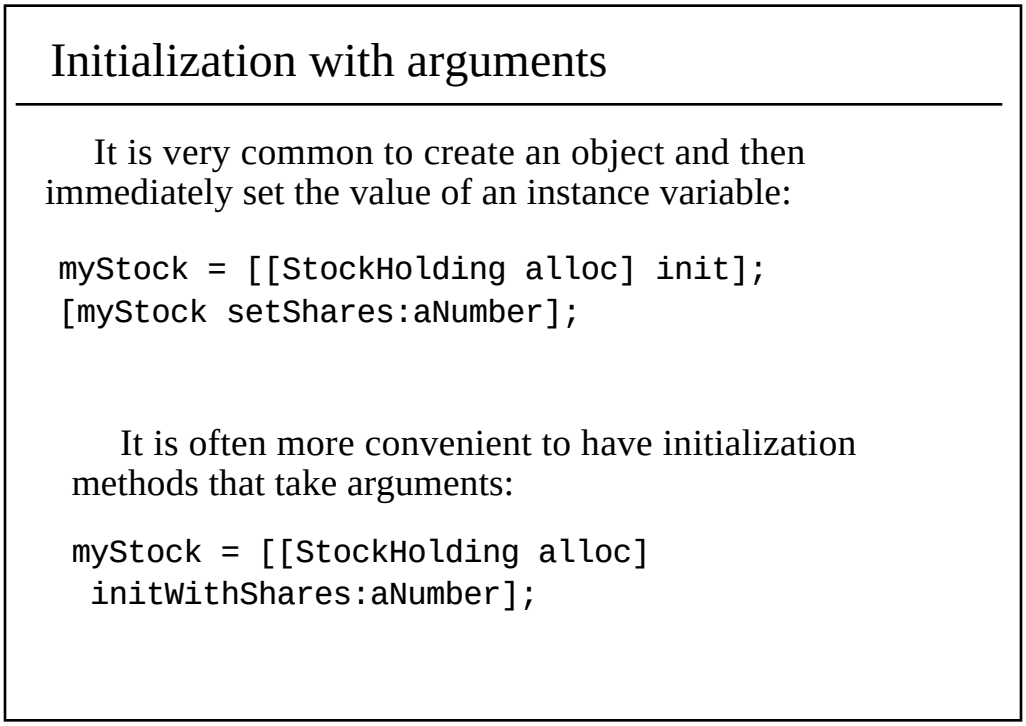
```
return self;
```



init in the class hierarchy

Suppose that the **init** method on the previous page is part of a class that is very low in a deep class hierarchy. Notice that although the subclass only calls its superclass's **init**, that **init** calls its superclass's **init** method. This continues all the way up to NSObject's **init** method. Then each class initializes its instance variables as the flow of control comes back down the hierarchy.

Thus, each class is only responsible for initializing the instance variables it declares. Inherited instance variables are initalized by the class that introduced them. For example, Asset's **init** method is responsible for initializing the instance variable **owner**. StockOption's init method doesn't have to initialize **owner**, but does have to initialize **expiration**.



Initialization with arguments

It's very common to have initialization methods that take arguments. If you have an instance variable called **shares**, you might have an initialization method that takes an initial value for **shares**. This method would be called **initWithShares:**.

Asset's initialization methods

Suppose you have a subclass of NSObject called Asset. Asset declares a single instance variable called **owner**. For an instance of Asset to be usable, you must set **owner** to a reasonable value.

So, you write a method called **initWithOwner:** that takes the owner of the asset being initialized as an argument:

```
- (id)initWithOwner:(Trader *)aTrader
{
   [super init];
   [self setOwner:aTrader];
   return self;
}
```

initWithOwner: is called like this:

```
myAsset = [[Asset alloc] initWithOwner: myTrader];
```

Of course, Asset inherits the method init from NSObject. So someone might do this:

```
myAsset = [[Asset alloc] init];
```

If you don't override **init**, an instance of Asset might be created that doesn't have a valid value for **owner**. You must override **init** in Asset to ensure that **owner** gets set to a usable value:

```
- (id) init
{
   Trader *defaultOwner;

   defaultOwner = [Trader floorBoss];
   return [self initWithOwner:defaultOwner];
}
```

Asset's **init** method gets a default value for **owner** and calls Asset's **initWithOwner:** method to do the initialization. Because **initWithOwner:** is the method that actually does the work, it's called the *designated initializer*.

StockOption's initialization methods

If you create a subclass of Asset called StockOption, it might have the instance variable **expiration**. As such, you'd create an initialization method called **initWithOwner:expiration:**.

```
- (id)initWithOwner:(Trader *)aTrader
         expiration:(NSCalendarDate *)aDate
{
   [super initWithOwner:aTrader];
   [self setExpiration:aDate];
   return self;
}
```

Notice how **initWithOwner:expiration:** calls the superclass's **initWithOwner:** method, instead of init. When calling a superclass's initializer, you should always call the superclass's designated initializer.

If you write a class with several initializers, be certain to clearly document which is the designated initializer. Generally, the designated initializer is the one that takes the most arguments.

Just as Asset had to override init, StockOption has to override **initWithOwner:** so the following call properly initializes **expiration:**

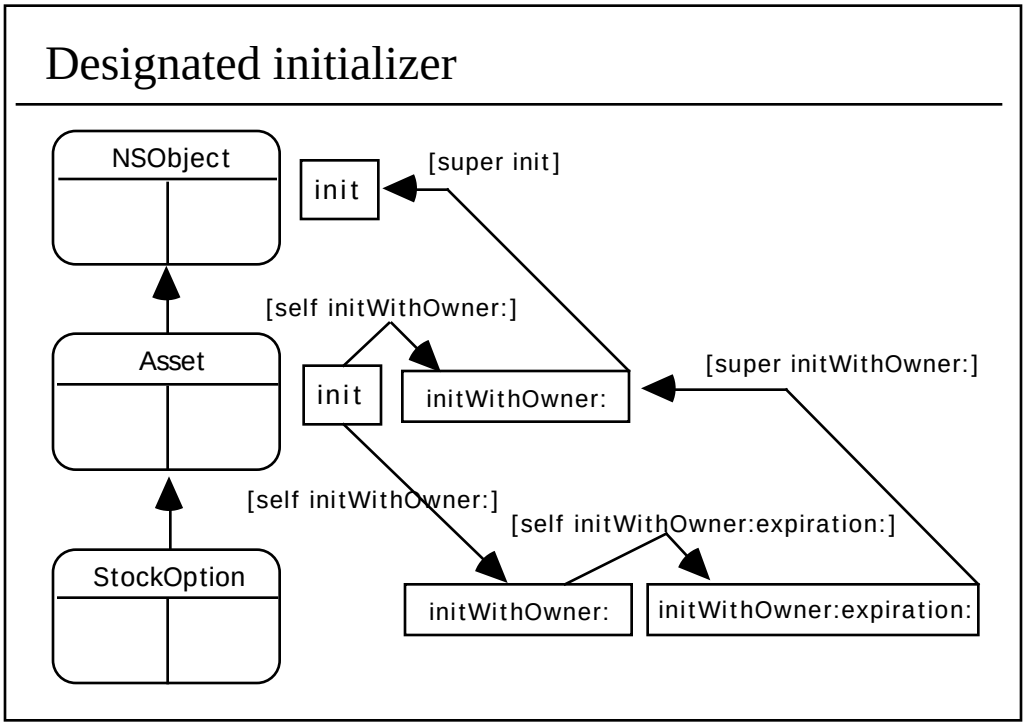
```
myOption = [[StockOption alloc] initWithOwner:
myTrader];
```

StockOption's **initWithOwner:** would probably look something like this:

```
- (id)initWithOwner:(Trader *)aTrader
{
   NSCalendarDate *now;

   now = [NSCalendarDate date];
   return [self initWithOwner:aTrader expiration:now];
}
```

StockOption doesn't have to override **init**.



Designated initializer

Why did **initWithOwner:expiration:** call Asset's **init** method? After all, if you call Asset's **init** method, it calls **initWithOwner:** for you.

The problem is, Asset's **init** method ends up calling **initWithOwner:** on **self**. **self** is a real variable. When an instance of StockOption executes Asset's **init** method, **self** still points to the instance of StockOption. So a message to **self** for the method **initWithOwner:** ends up executing StockOption's **initWithOwner:** method. StockOption's **initWithOwner:** calls **initWithOwner:expiration:**, and you've got an infinite loop.

Moral—in the designated initializer, call the superclass's designated initializer.

Why didn't you have to override **init** in StockOption? Suppose someone sent init to an instance of StockOption. StockOption doesn't have an **init**, so the **init** method inherited from Asset is used. Asset's **init** calls **initWithOwner:** on **self**. In this case that is StockOption's **initWithOwner:**, because **self** is an instance of StockOption. Thus **expiration** gets initialized properly.

Moral—it's only necessary to override the superclass's designated initializer.

Rules for initialization methods

1. If a class has any initialization methods, it must have one designated initializer.
2. All initialization methods except the designated initializer must call the designated initializer, either directly or indirectly.
3. The designated initializer of a subclass must call the designated initializer of its superclass.
4. A subclass must override its superclass's designated initializer.
5. Class documentation must clearly indicate which initialization method is the designated initializer.

dealloc

```
- (void)dealloc
{
  [writer release];
  [expiration release];
  [company release];
  [super dealloc];
}
```

dealloc

If you allocate or retain any objects in an initialization method, you must release them in the **dealloc** method. Then call the superclass's **dealloc** method.

Remember that **dealloc** is called when an object's retain count becomes zero. If you forget to release objects that you created, they will not be cleaned up, and your program will leak memory.

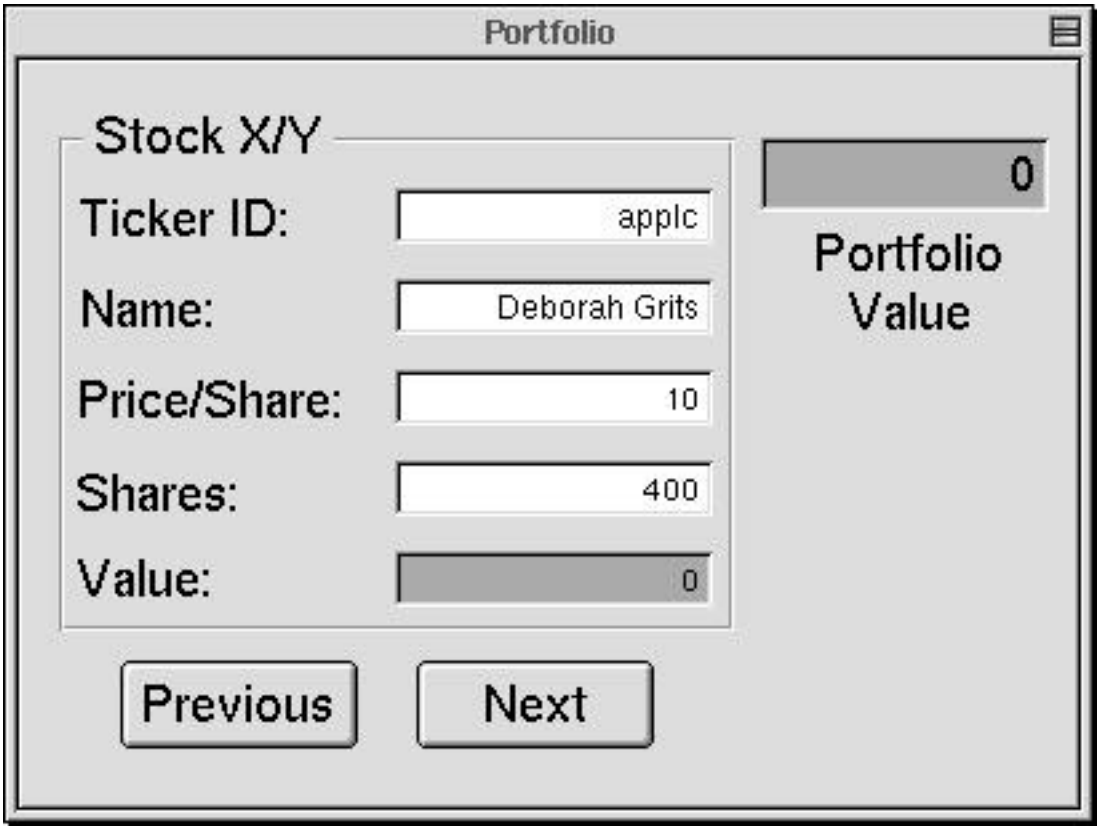
You should not release inherited instance variables. The superclass's **dealloc** method takes care of releasing them. Just as with initialization methods, you're only responsible for instance variables that you declare.

Abstract classes

A class is abstract if it is not used to make direct instances, but only used as a base from which other classes inherit. For example, NSObject is an abstract class. You'd never create an instance of NSObject, but all of your classes inherit from NSObject.

Abstract classes define behavior that all their subclasses inherit. They're generally use to factor out common behavior from a group of related classes. For example, real estate, stock holdings, and stock options are all different kinds of assets. They each have their own custom behavior, but there is also behavior that's common to all three. By grouping them in a class hierarchy under the abstract class Asset, you avoid having to reimpliment the same behavior in three different places. This makes it much easier to maintain.

DEMONSTRATION 6.1 PORTFOLIO MANAGER: USER INTERFACE TEMPLATE



The Portfolio Manager application isn't very interesting yet. All it does is notify you before its window closes and before it quits, and allow you to cancel either action.

To make Portfolio Manager a more functional application, it needs to be able to display one stock holding in a portfolio and switch between multiple stock holdings in a portfolio. To avoid getting involved with the details of setting up the user interface, the source code for a user interface is available in the exercise materials directory. In this demonstration you build the provided template project and examine its source code. In future exercises, you will add functionality to the application by inserting your own code into the project.

Objectives

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

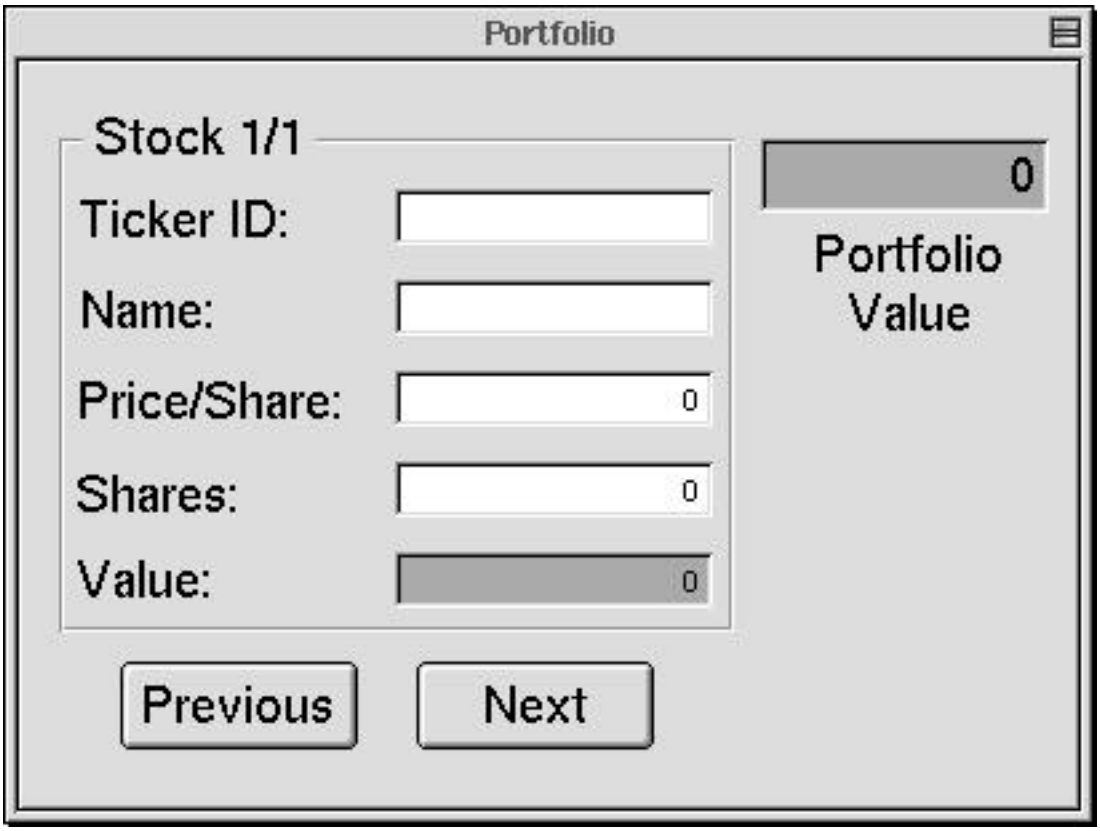
- Explore an existing application to figure out how it works
- Identify provided classes that are suitable for subclassing
- Locate methods in PortfolioController where you can add code to extend the functionality of Portfolio Manager

Demonstration

1. Copy 06.1_PortfolioManager from ExerciseMaterials/Exercises. Open the project in Project Builder.

2. Build and run Portfolio Manager. It presents you with a single window and some menu items.

3. Choose the New Stock command in the Stock menu. This creates a new stock holding, displayed in Portfolio Manager's main window:



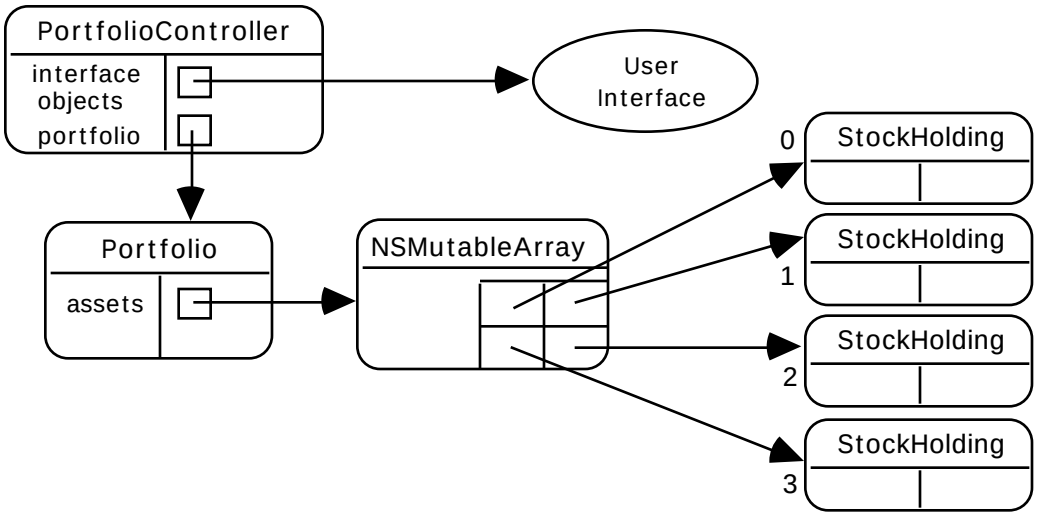
4. Choose New Stock again to create another stock holding.

5. Use the Next and Previous buttons to switch between the two stock holdings. Try to edit information in the different fields. What information is stored for each stock holding? What information is only stored in the user interface objects?

6. Delete the first stock holding. Choose the Delete Stock command in the Stock menu.

7. Delete the remaining stock holding. What happens when you choose the Delete Stock command again?

8. Now examine the object model. At a high level, Portfolio Manager's object model looks like this:



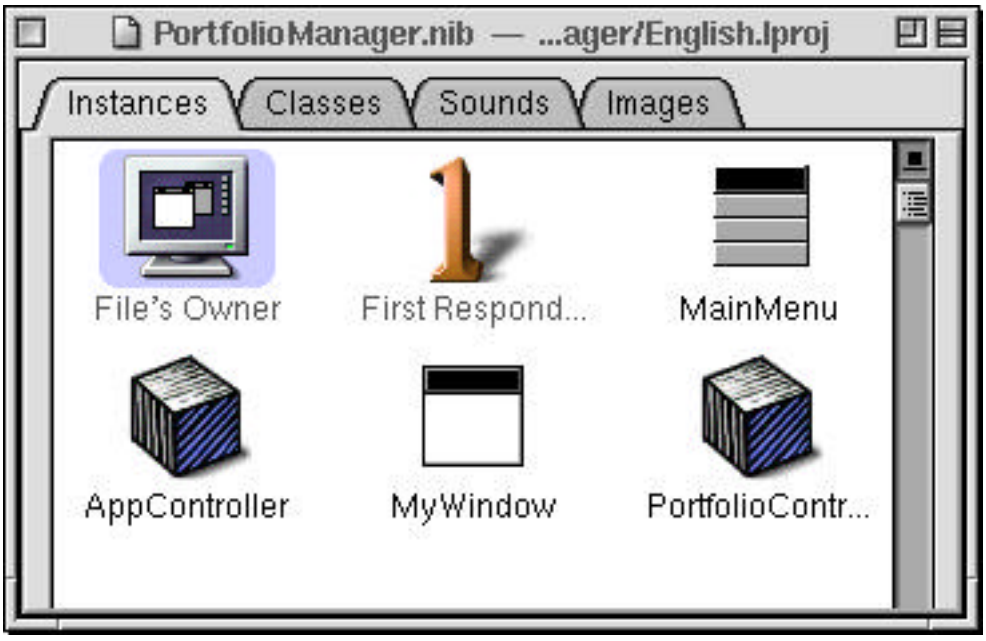
A single Portfolio object is used to manage a collection of Assets. Portfolio uses an NSMutableArray to assist it. In this application, the Assets are instances of StockHolding, a subclass of the abstract Asset class. These objects are the model of a portfolio in the Model-View-Controller design pattern.

The user interface consists of a window with text fields and buttons that display a single stock holding at a time and allows the user to modify the attributes of the stock holding. There is also a user interface to add, delete, and switch between stock holdings. These objects comprise the view.

An instance of PortfolioController ties the information in the model to the user interface. This object is responsible for getting information from the model objects and putting it into the view when a new stock holding is displayed. It is also responsible for propagating changes from the user interface back to the model objects. PortfolioController is the controller for a portfolio.

9. Switch back to Project Builder and open Portfolio Manager's interface file. Because Interface Builder is such a powerful tool, a lot of information can be stored in an interface file. One of the first steps towards understanding how an application works is looking at its interface file and tracing some of the connections.

   The first task is to identify the controller object for the application. Looking at the File window, it's clear that PortfolioController is the main controller for the application.



10. Select the PortfolioController instance and bring up the Connections inspector. Notice that the PortfolioController has access to the text fields in the user interface.

11. Select the Next and Previous buttons. The Connections inspector reveals that both buttons target PortfolioController, and each is set to send an appropriate message.

12. Finally, look at the Stock menu. Again, examine the connections for each of the two menu items.

13. Now take a look at the classes used in Portfolio Manager. Portfolio Manager uses two provided library classes, Portfolio and Asset. Switch to Project Builder and look at the **Asset.h** header file.

    Asset is an abstract superclass. It keeps track of the name of an asset and provides supporting methods for initialization, deallocation, and archiving. Archiving is discussed in Chapter 14, Archiving. Asset also declares a method, **value**, that returns a double. Subclasses of Asset are expected to override this method. They should provide an implementation that calculates the value of the asset in dollars and returns it as a double.

    Notice the @class entry. @class is a compiler directive that notifies the compiler that the following entries are class names. Asset's interface file doesn't need to know anything about NSString except that it's a class name. Because interface files are frequently imported by other classes, it speeds compilation to have interface files import as few other interfaces as possible. The @class directive makes it possible to wait until the implementation file to import interfaces other than your superclass.

14. Look at the Portfolio.h header file.

    A Portfolio keeps track of a list of Assets, allowing the user to add, delete, or access Assets by index. It also implements a **value** method that calls the **value** method of each Asset in turn, adding the results together to return the total value of the Portfolio.

15. Examine the header and implementation files for the StockHolding class. Notice that this class is a subclass of Asset but as of yet it has no additional instance variables or behavior. This is the class you'll write in the exercise.

16. Tying the model objects, Portfolio and its associated Assets, together with the user interface is a PortfolioController. Portfolio Manager has one instance of PortfolioController that manages a single Portfolio, displayed in the main window. Open PortfolioController's interface file in Project Builder. First come some #import statements.

    Because the interface to a controller object is not normally imported by other classes, it's acceptable to import the entire Application Kit precompiled header. However, there's still no need to import **Portfolio.h** in the header file. You can simply use the @class directive and import **Portfolio.h** in the implementation file.

17. Next comes the declaration of instance variables. PortfolioController declares a number of outlets for the different user interface elements it needs to know about. In addition, it has two instance variables that are not for user interface elements. **displayedIndex** keeps track of the index of the currently selected StockHolding, and **portfolio** provides a way to communicate with the Portfolio instance the PortfolioController manages.

18. The next set of instance methods manage the portfolio's stock holdings. **deleteStock:** removes the currently selected stock holding from the portfolio and resets **displayedIndex** to point to the next stock in the list. **newStock:** creates a new instance of StockHolding and adds it to the portfolio. It also resets **displayedIndex** to point to the newly created stock holding. **nextStock:** and **previousStock:** traverse the list of stock holdings.

19. The final set of instance methods coordinate changes to the user interface with changes to the object model. PortfolioController calls **saveChangesToSelection** when the selection is about to change. This method saves information from the user interface to the corresponding stock holding in the object model. **fillFields** takes information from the selected stock holding and puts it in the user interface. Finally, **blankFields** zeroes out the fields when there is no selection.

20. Open PortfolioController.m. Examine some of the implementation methods.

Important ideas from this lesson

» When an object receives a message, it looks for the implementation of the corresponding method by following the **isa** pointer to its class. If the method is not found in that class, it searches up the class hierarchy until it finds an implementation.

» A subclass can do any of the following:

Add instance variables

Add methods

Override methods

Extend methods using **super**

» A subclass is not allowed to do either of the following:

Remove inherited instance variables

Remove inherited methods

» **super** is used to call the inherited version of a method.

» An initializer method initializes instance variables.

» A designated initializer must do the following:

call the superclass's designated initializer

initialize instance variables that are not inherited

return **self**

» All non-designated initializers call the designated initializer, either directly or indirectly.

» If you write any initializers in a class, you must override the superclass's designated initializer.

» In **dealloc**, release any helper objects and call super's **dealloc**.

REVIEW INHERITANCE

1. How many superclasses can a class have? How many subclasses?

2. You are writing a program and have to create two classes: Building and House. Which is most likely to be the superclass of the other?

For the following questions imagine two classes: X and Y. X is a superclass of Y. X has an instance variable **phoneNumber** that refers to an NSString. Y has an instance variable **age** that is an int.

3. Both X and Y have implementations of the instance method **liquefy**. If you create an instance of Y and send it the message **liquefy**, will the implementation in Y or X get executed?

4. In X's **init** method, what instance variables must be initialized?

5. In the space below, write an **init** method for Y.

EXERCISE 6.1 PORTFOLIO MANAGER: STOCK HOLDING

Now that you understand the basics of Portfolio Manager's user interface and object model, it's time to extend that object model. In this exercise you create a subclass of Asset called StockHolding. A StockHolding has several instance variables not present in Asset, including a price and number of shares. Using this information, an instance of StockHolding can calculate its value. All of this additional information should be accessible through the user interface.

Objectives

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

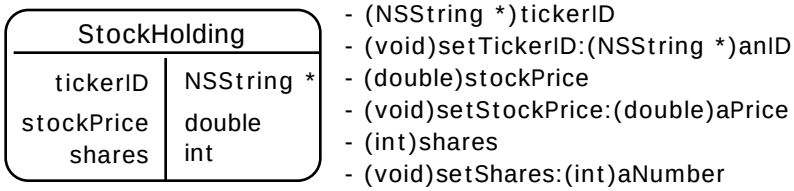
- Add instance variables to a subclass
- Correctly override an inherited **init** method
- Add methods to a subclass
- Implement a method declared in an abstract superclass
- Add code to an existing controller class to support new functionality

Exercise

1. Open the Portfolio Manager project.

2. StockHolding adds support for several instance variables not declared in Asset. Specifically, a stock holding has an NSString to represent its ticker ID, a double to store its stock price, and an integer to store the number of shares. Add declarations to **StockHolding.h** for these instance variables.

3. Add declarations for accessor methods for each new instance variable you added to StockHolding.



4. Write implementations for the accessor methods.

5. Override **init** to set the new instance variables to reasonable initial values. Use the accessor methods you wrote!

6. Override the **dealloc** method inherited from Asset. You need to release any objects used by StockHolding—any objects instances of StockHolding refer to via an instance variable.

7. Asset declares a **value** method that subclasses are supposed to implement. Override **value** and make it calculate the value of the stock holding based on the price of the stock and the number of shares.

8. Build the project to make sure you don't get any compiler errors. Building a project after completing work on a class means you don't have to try to fix code in two different classes at once.

9. Run Portfolio Manager. Try to set the price and number of shares for some stock holdings. What happens when you switch to a different stock holding and then come back?

10. Portfolio Manager doesn't save the changes you make in the user interface to the selected StockHolding instance. To fix this, you need to make some changes to PortfolioController. Add code to **saveChangesToSelection** and **fillFields** that coordinates the new instance variables in StockHolding with the user interface.

11. Build and run Portfolio Manager. Test to make sure you can save the stock price, number of shares, and ticker ID for stock holdings. Does Portfolio Manager update the value field immediately when you change the number of shares or stock price? Does it correctly display the value after you switch to a different stock and come back? Speculate on why the user interface works this way.

Enhancements

» Modify PortfolioController to disable the Next and Previous buttons when they won't do anything. Make sure you enable the appropriate button when a new stock is created, and disable the appropriate button or buttons when a stock is deleted.

» Create an **awakeFromNib** method in PortfolioController where you blank out the fields. Why does this code have to go here instead of in PortfolioController's **init** method?

» Read the documentation on NSTextFieldCell to see what notifications it posts. Set up PortfolioController as an observer of notifications from the stock price and shares text fields so you can update the value fields on the fly, instead of just when the fields get filled after a next, previous, new, or delete command.

» Add an **insertStock:** method to PortfolioController that inserts the new stock at the current location in the list of stocks, instead of at the end. Watch out for edge cases! Make sure your method works when the currently selected stock holding is the last one, as well as when there are no stock holdings in the portfolio.