Chapter 4

Creation and Destruction of Objects

CHAPTER 4 CREATION AND DESTRUCTION OF OBJECTS

Goal

To understand how objects are created and destroyed.

Prerequisites

A basic understanding of C's malloc function.

Objectives

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

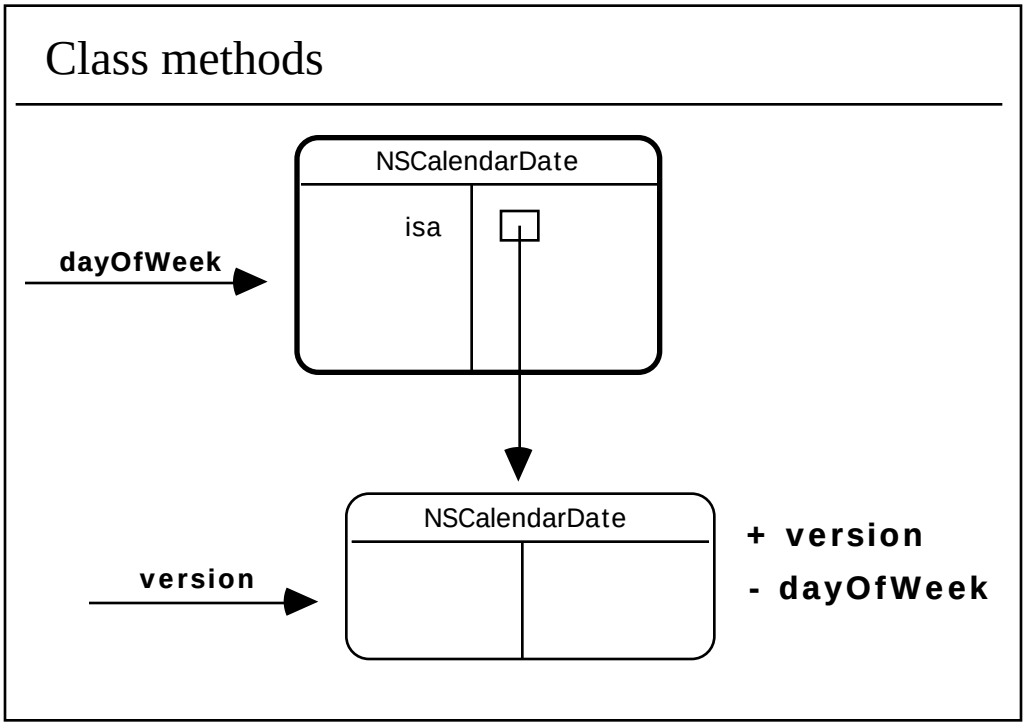
- Write an application that programmatically creates and destroys objects
- Properly use **retain**, **release**, and **autorelease**

Reading

These references contain more information about creation and destruction of objects.

/System/Documentation/Developer/ReleaseNotes/Foundation

/System/Documentation/Developer/TasksAndConcepts/ ObjectiveC (The Objective-C Language.)



Class methods

A class is not an instance itself. It has no instance variables of its own and it can't perform methods intended for instances of the class. However, a class definition can include methods intended specifically for the class object—**class methods** as opposed to instance methods. A class object inherits class methods from the classes above it in the hierarchy, just as instances inherit instance methods.

In source code, the class object is represented by the class name. In the example, the NSCalendarDate class returns the class version number using a method inherited from NSObject:

```
int versionNumber = [NSCalendarDate version];
```

Declarations of instance methods begin with "-". Class methods have declarations beginning with "+". Because classes are sometimes referred to as factories, class methods are sometimes called "factory methods."

```
+(int)version          // this is a class method

-(int)dayOfWeek        // this is an instance method
```

alloc and init

```
Employee *theEmployee;
id purchaseDate;
NSArray *employees;

theEmployee = [[Employee alloc] init];

purchaseDate = [[NSCalendarDate alloc] init];

employees = [[NSArray alloc]
               initWithObject:theEmployee];
```

alloc and init

A principal function of a class object is to create new instances. This code tells the NSCalendarDate class to create a new NSCalendarDate instance and assign it to the **purchaseDate** variable:

```
NSCalendarDate *purchaseDate;
purchaseDate = [NSCalendarDate alloc];
```

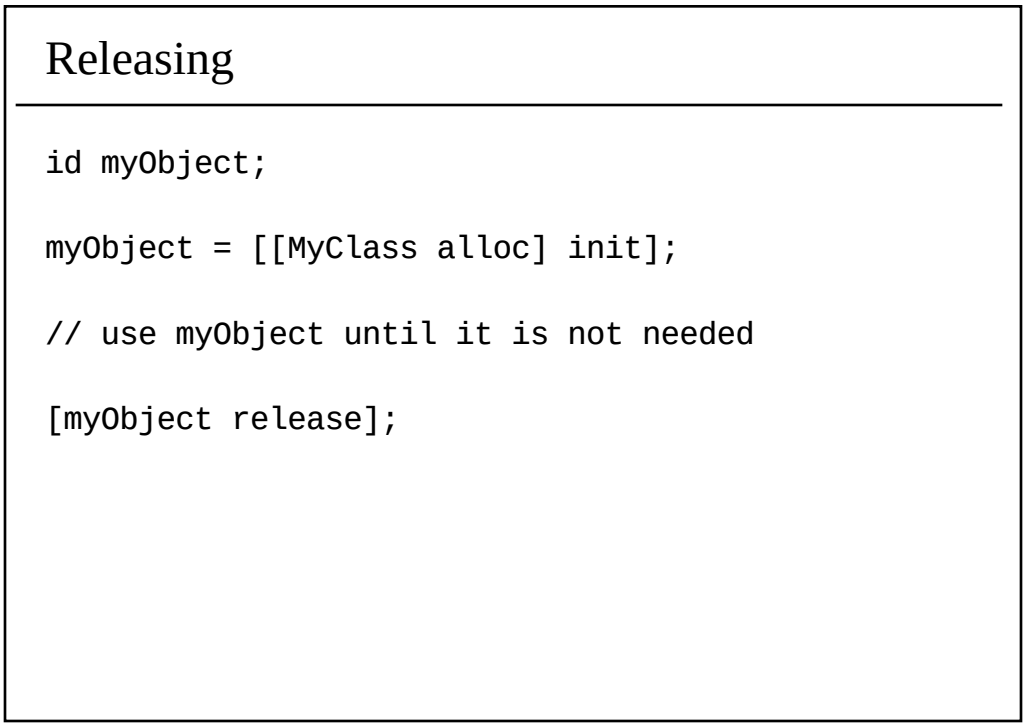
The **alloc** method dynamically allocates memory for the new object's instance variables. This is much like the C function malloc.

alloc also initializes all the instance variables to 0. All, that is, except the **isa** variable that connects the new instance to its class. For an object to be useful, it generally needs to be more completely initialized. That's the function of an **init** method. Initialization typically follows immediately after allocation:

```
purchaseDate = [[NSCalendarDate alloc] init];
```

This line of code, or one like it, would be necessary before **purchaseDate** could receive any of the messages illustrated in previous examples. The **alloc** method returns a new instance and that instance performs an **init** method to set its initial state.

Initialization methods often take arguments to allow particular values to be passed and have keywords to label the arguments, but they all begin with "init." **initWithYear:month:day:hour:minute:second:timeZone:**, for example, is a method that initializes a new NSCalendarDate instance.

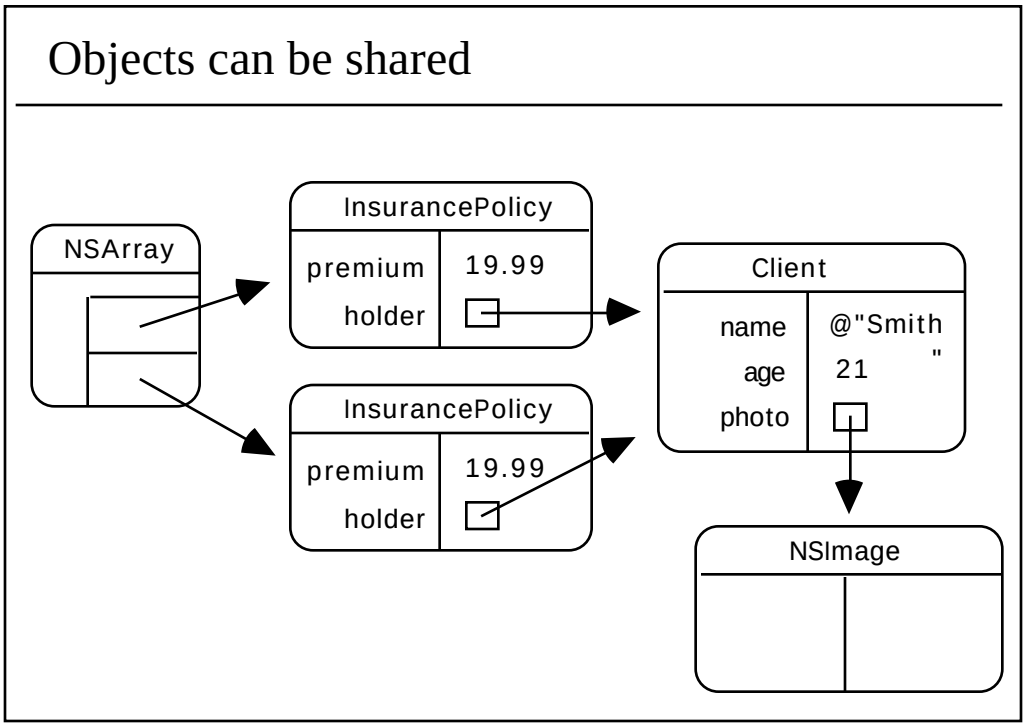


Releasing

In C, whenever you malloc some memory, it's important to free that memory when you no longer need it. This increases the performance of your system over the long run. If you forget to free memory allocated with malloc, your program is said to be "leaking."

The same is true for **alloc**. If you create an object using **alloc**, it is crucial to release the object when you no longer need it. For example:

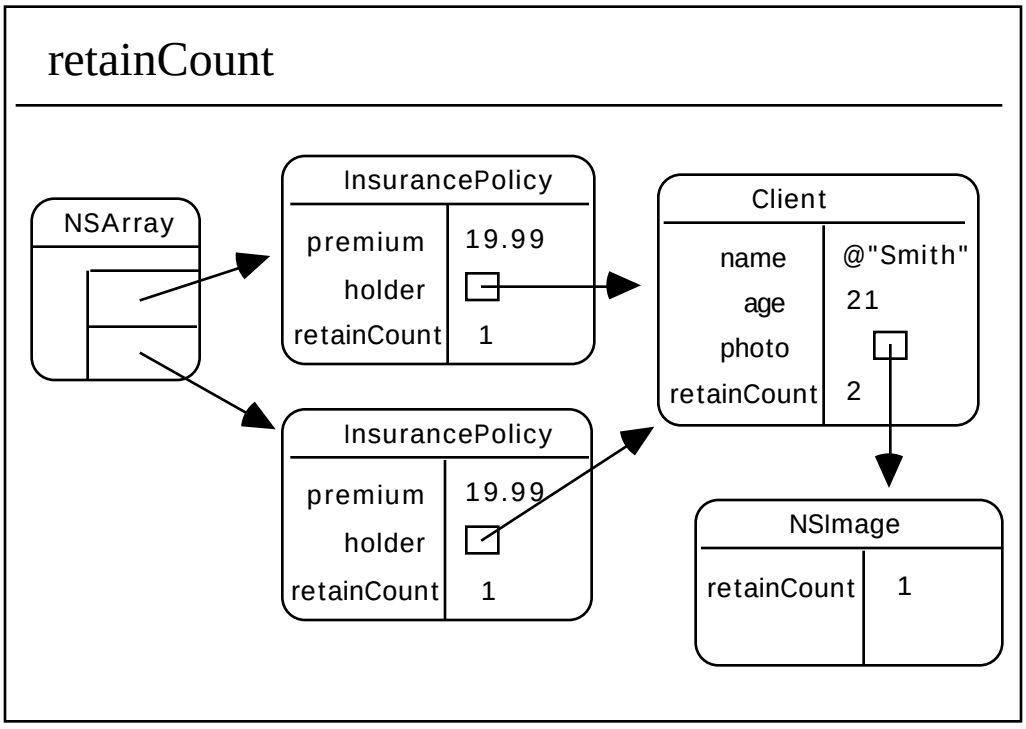
```
id myObject;

myObject = [[MyClass alloc] init];

// use myObject until it is not needed

[myObject release];
```



Objects can be shared

Objects can be shared. For example, if one client buys two life insurance policies, each policy will have a reference to the same Client object. If one Insurance policy is deleted, the Client should remain. If the second policy is deleted, the Client should be deleted.

This poses a problem: Who's responsible for keeping track of a shared object?



retainCount

To deal with this problem, each object keeps track of how many other objects are "retaining" it. For example, a retain count of 2 for a Client object indicates that 2 different objects are retaining it.

When the retain count becomes zero, no one is retaining the object. The object can then be deallocated. This answers the question of who is responsible for deleting objects. The retain count mechanism keeps track of how many objects care about each object in an application. When no one cares about an object, it is removed from memory.

retain and release

retain and release modify the retain count

- retain increments the retain count
- release decrements the retain count

dealloc deallocates the memory of an object

- when the retain count reaches 0, the object is sent a dealloc message

retain and release

To make sure an object's retain count reflects how many objects are using it, your objects must retain objects they use.

To retain an object, send it the **retain** message. **retain** increments the retain count. Objects that you create using **alloc** or **copy** are automatically retained, so you don't have to retain them a second time.

To release an object you previously retained, send it a **release** message. **release** decrements the retain count. Only release objects you previously retained, either explicitly by sending them a **retain** message, or implicitly by creating them using **alloc** or **copy**.

When an object's retain count reaches zero, the object is sent a **dealloc** message. This deallocates the memory used by the object. You should never call **dealloc** yourself—only call **release**.

Pitfalls using release

One of the problems of using **release** is that when an object's retain count becomes 0, the object is deallocated. This causes problems if you're not careful.

For example, consider NSCalendarDate's **description** method. Internally, NSCalendarDate keeps track of the date and time using some large number, or potentially several numbers. NSCalendarDate's **description** method returns a nicely formatted string that describes the date. NSCalendarDate probably doesn't have an NSString as an instance variable—rather, it generates one on the fly, when someone requests the description.

Here's one possible implementation of **description**:

```
- (NSString *)description
{
   NSString *desc;

   desc = [[NSMutableString alloc] init];
   // somehow change the description string to
   // a text description of the current date
   return desc;
}
```

Unfortunately, this method leaks an NSString—it allocates an NSString, but doesn't release it.

What about this:

```
- (NSString *)description
{
   NSString *desc;

   desc = [[NSMutableString alloc] init];
   // somehow change the description string to
   // a text description of the current date
   [desc release];
   return desc;
}
```

This doesn't work either, because as soon as you call release, the string goes away. desc is now a pointer to a freed object, and therefore invalid. There must be some other way of telling an object to go away without having it go away immediately.

autorelease

autorelease marks an object to be sent release sometime in the future

```
- (NSString *)description
{
    NSString *desc;

    desc = [[NSMutableString alloc] init];
    // somehow change the description string to
    // a text descripttion of the current date
    [desc autorelease];
    return desc;
}
```

autorelease

The problem on the previous page is corrected by using **autorelease**. Sending **autorelease** to an object means that object will be sent release sometime in the future. The object stays around until the end of the event loop.

Often you ask one object for another. For example, you might ask a text field for its string. It creates an instance of NSString and returns it to you. You should assume the string is autoreleased. It will disappear unless you retain it.

The following messages are the only exceptions:

» **alloc**

» **copy**

Objects created using **alloc** and **copy** have a retain count of one and have not been autoreleased. Objects returned by any other method are autoreleased, and should be retained if you want them to stay around.

This includes “convenience methods” that return newly created objects.
For example, NSCalendarDate has a **date** method that returns a new NSCalendarDate initialized to the current date. Even though this method creates a new instance of NSCalendarDate, the return value does not come from **alloc** or **copy**. Therefore the object should be considered autoreleased.

Accessor methods

It is important that accessor methods autorelease and retain objects correctly. When someone sets an instance variable for your object, you must autorelease the old object and retain or copy the new one. Here is the set method for the instance variable **photo**, used to refer to an NSImage object:

```
- (void)setPhoto:(NSImage *)anImage
{
   if (photo != anImage) {
      [photo release];
      photo = [anImage retain];
   }
}
```

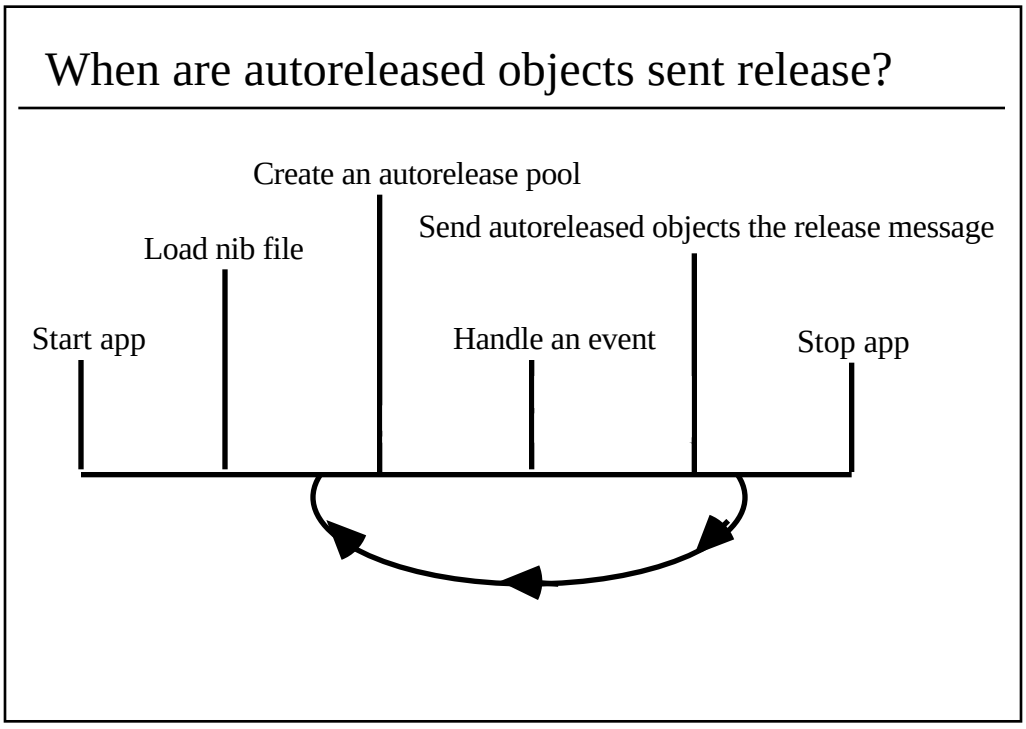
Notice the if clause. What would happen if someone had a pointer to **photo** and sent it as the argument to **setPhoto:**? Without the if clause, **setPhoto:** would send a **release** message to **photo**—its retain count would go to 0, and **photo** would be deallocated. There would be no new image to retain, because **photo** and **anImage** pointed to the same object. The if clause prevents this problem.

Assuming the new image is different, the old image is released. The new image is retained. The instance variable **photo** is assigned the address of the new image. In this way, memory used by the old image is reclaimed, and the new image stays around until you no longer need it.

You have a choice when you write an accessor method. You can choose to retain the object sent in as the argument, or you can create a copy of that object. Generally, you should copy objects that can be considered simple data-bearing or value objects. You should retain business objects, or other objects that your object has a relationship to.

Examples of value objects are NSString and NSCalendarDate. Strings and dates are treated as objects for convenience, but their only behavior is to modify their data. They're not business objects. Your object doesn't have a relationship with a string. Rather, your object uses a string to store one of its attributes.

With business objects, your object generally has a relationship with the business object. For example, an employee has a manager. The employee doesn't use the manager object to store the name of its manager. Rather, the manager object is another full-fledged employee object in its own right. In this case, you should retain the object instead of copying it.



When are autoreleased objects sent release?

Before an event is fetched from the queue, an autorelease pool is created. An event is then fetched from the queue. Some code gets executed as a result. In the code, whenever **autorelease** is sent to an object, the object is added to the autorelease pool. The autorelease pool is basically a list of autoreleased objects. When the event has been handled, every object in the pool is sent a **release** message.

Notice that all autoreleased objects are destroyed between events. There is nothing going on between events, so none of the autoreleased objects are being used.

Rules for Managing Memory

1. Objects created by **alloc** or **copy** are automatically retained.
2. All other objects are considered autoreleased.
3. If you need to keep an autoreleased object, retain it.
4. When you no longer need an object, release it.
5. Objects are automatically retained when added to a collection.
6. Objects in a collection are released when the collection is deallocated.
7. Never send **dealloc**, it's called automatically.
8. Never release an object unless you retained it.
9. Use **autorelease** for newly created objects used as return values.

Convenience methods

Some objects are frequently initialized in the same way and/or created and used for brief periods of time. For example, you frequently want an NSCalendarDate object that represents the current date and time, and often you only want that object for a single comparison.

To make it easier to get objects of this sort, many classes provide convenience methods for creating instances. A convenience method that creates an object is a method that allocates and initializes the object for you according to some preset parameters. Because you did not call **alloc** to create them, objects created by convenience methods should be considered autoreleased.

For example, consider the following code:

```
NSArray *x, *y;
id z;

z = [[[Employee alloc] init] autorelease];
x = [[NSArray alloc] initWithObject:z];
y = [NSArray arrayWithObject:z];
```

Both the assignment to **x** and the assignment to **y** do the same basic operation. Each creates an array with a single member object, **z**. However, in the first case the array is created using **alloc**. Therefore, the object **x** has not been autoreleased and will not go away until sent a **release** or **autorelease** message. **y** was created using a convenience method. It has been autoreleased, and will go away after the event loop is over unless someone retains it.

Notice how **z** was created. It's fairly common to allocate an object and then immediately autorelease it if you plan to add the object to a collection class, because collection classes retain the objects in them.

DEMONSTRATION 4.1 DOCUMENTATION

Class documentation is accessible through ProjectBuilder's Find panel. This demonstration gives a brief introduction to using the Find panel and describes some of the additional documentation resources available to you.

Objectives

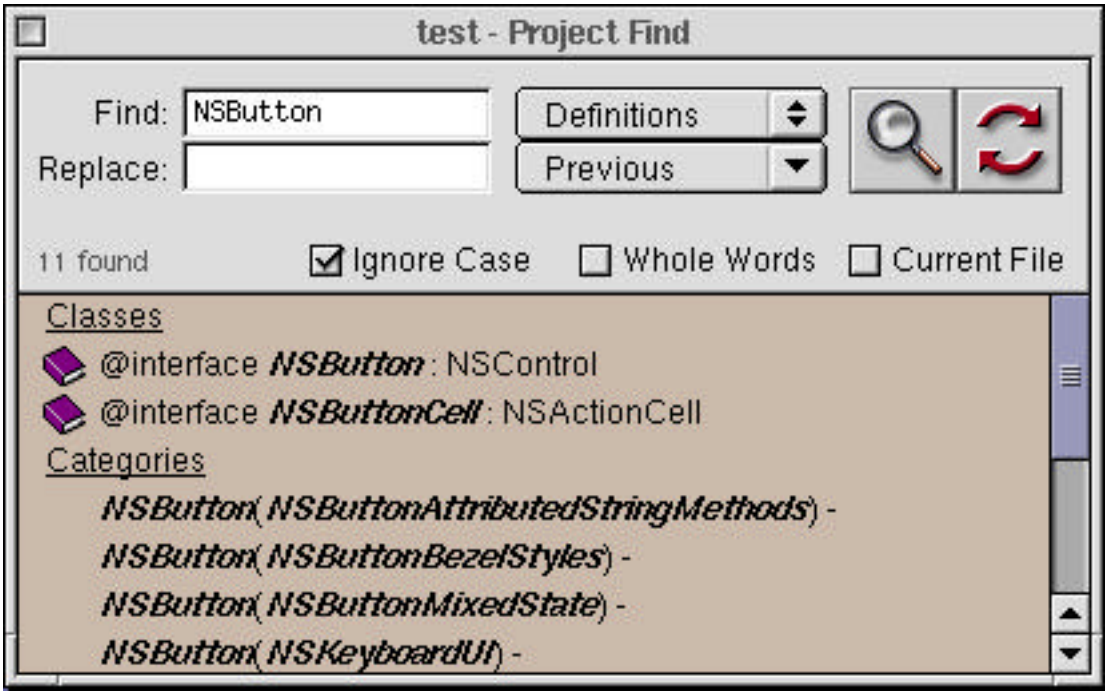
After completing this demonstration, you'll be able to:

- Find framework reference materials using ProjectBuilder's Find panel.
- Read a class specification sheet.

Demonstration

1. Bring up ProjectBuilder's Find panel by clicking on the Project Find button in the project window (it has a magnifying glass icon).

2. Type NSButton in the find text field and push return.



3. The Project Find panel searches for definitions and displays the search results. Clicking on any of the book icons will cause the class reference to be displayed. Clicking on any of the text will cause the header file definition to be displayed.

   Click on the book icon for NSButton.

4. All class documentation has a common format.

   The first thing listed is the class's superclass (in the Inherits From: section), what protocols the class conforms to, and what header file it's declared in.

   Next there's a class description, giving a general overview of the class and some suggestions as to how to use it.

   Next, all the methods declared in the class are listed; you can click on any of the methods for an explanation of what the method does.

Important ideas from this lesson

- You can send messages to classes. The methods they invoke are class methods.
- **alloc** is a class method that creates instances of the class.
- **init** is an instance method that initializes the instance variables of the object.
- Each object keeps track of how many other objects are retaining it.
- If the retain count of an object reaches zero, the object is sent **dealloc**.
- **dealloc** frees the object.
- Accessor methods must release old values and retain new ones.
- Autoreleased objects are sent the **release** message between events.
- Objects that inherit from NSObject conform to rules for memory management. Keep these rules in mind when programming.

REVIEW CREATION AND DESTRUCTION

1. What is a class method?

2. How would you invoke a class method?

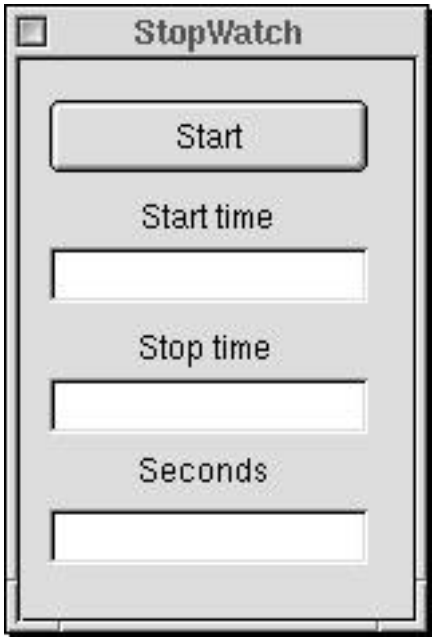
3. What does alloc do?

4. What does init do?

5. Circle the cases where the object should be considered autoreleased:

```
myEmployee = [[Employee alloc] init];

purchaseDate = [NSCalendarDate calendarDate];

name = [myTextField stringValue];

newName = [name copy];

newName = [[name copy] autorelease];
```

EXERCISE 4.1 STOPWATCH



The past two exercises have shown the basics of using Project Builder and Interface Builder to create an application. So far, the applications have been heavy on connections in Interface Builder, and light on source code.

In this exercise, you create an application that calls methods of some ApplicationKit classes programmatically. Just as Interface Builder can set the action method of an NSButton, so can an object you write. Using the example of a stop watch with a single button, this exercise explores some of NSButton's methods. It also introduces a class from Foundation—NSCalendarDate.

Objectives

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

- Programmatically set the action of a NSButton
- Use NSCalendarDate objects to record times
- Display times in a text field

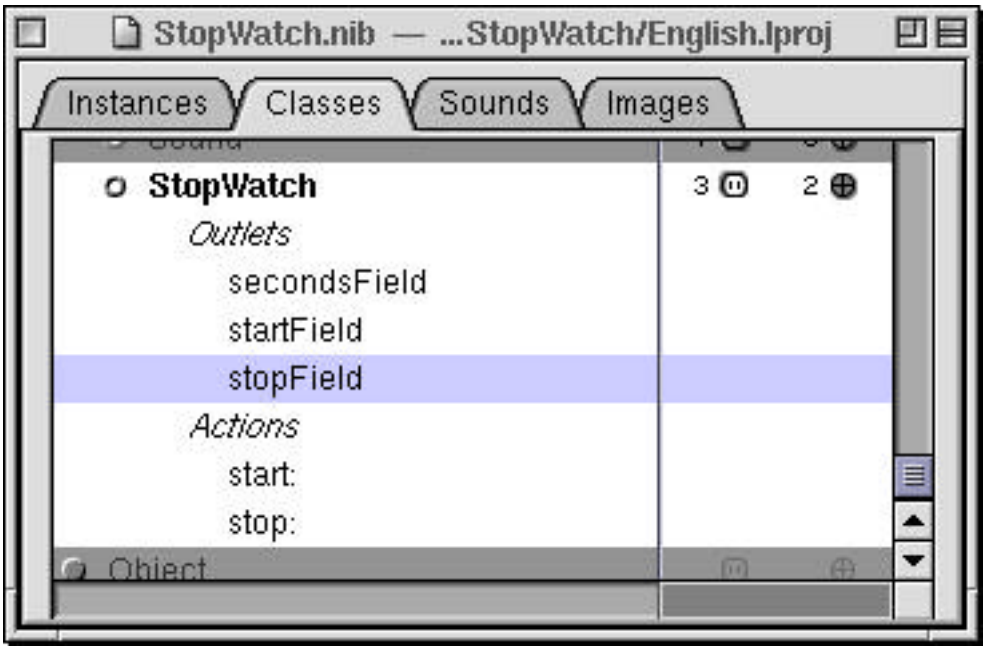
Exercise

1. Create a new application project in Project Builder. Name your application StopWatch.

2. Create the user interface. Use Interface Builder to set up a user interface with a single button and three text fields, as shown in the illustration.

3. Create a StopWatch controller object using Interface Builder. Switch to the classes view. Select NSObject and choose Subclass from the Classes menu item. Name the new subclass StopWatch.

4. Add appropriate outlets and action methods to the StopWatch class. You add outlets by selecting the small plug icon and typing return. You add action methods by selecting the small cross hairs icon and typing return.



   To decide what action methods the StopWatch needs, consider what its functionality is going to be. This application models a stop watch with a single button. When a user presses the button once, the stop watch resets itself to zero and starts running. When the user presses the button again, the stop watch stops running and displays the elapsed time.

   The user interface for the StopWatch application records the start, end, and elapsed time. In order to display these values, the StopWatch will need access to the appropriate text fields, in other words, outlets.

5. Create an instance of StopWatch by choosing the Instantiate command in the Classes menu.

6. Connect the objects in your user interface to the StopWatch object. What objects does the StopWatch know about? What action should the Start button send?

7. Create skeleton .m and .h files for the StopWatch class by selecting the StopWatch object and switch back to the Class view. Choose Create Files from the Classes menu.

8. Write the code for StopWatch class. StopWatch has two main methods, **start:** and **stop:**. Each method has a number of tasks to accomplish:

   - Allocate an NSCalendarDate and store it for later use
   - Change the title of the button from Start to Stop or viceversa
   - Change the action of the button to match the title
   - Update the user interface

   You can send messages to the button by messaging the **sender** variable passed in as the argument of your action method. Look at the documentation of NSButton's **setAction:** and **setTarget:** methods. You'll also want to check NSCalendarDate's documentation—you'll find the class method **calendarDate** useful.

   Hint: You can get the selector for a method name using the @selector compiler directive. Use the method **timeIntervalSinceDate:** to determine the number of seconds between two dates. This method is documented under NSDate, the superclass of NSCalendarDate.

9. Build and test your application.