Object Ownership and Automatic Disposal

In an Objective-C program, objects are constantly creating and disposing of other objects. Much of the time an object creates things for private use and can dispose of them as it needs. However, when an object passes something to another object through a method invocation, the lines of ownership—and responsibility for disposal—blur. Suppose, for example, that you have a Thingamajig object that contains a number of Sprocket objects, which another object accesses with this method:

– (NSArray *)**sprockets**

This declaration says nothing about who should release the returned array. If the Thingamajig object returned an instance variable, it's responsible; if the Thingamajig created an array and returned it, the recipient is responsible. This problem applies both to objects returned by a method and objects passed in as arguments to a method.

Ideally a body of code should never be concerned with releasing something it didn't create. The Foundation Framework therefore sets this policy: *If you create an object (using **alloc** or **allocWithZone:**) or copy an object (using **copy**, **copyWithZone:**, **mutableCopy**, or **mutableCopyWithZone:**), you alone are responsible for releasing it.* If you didn't directly create or copy the object, you don't own it and shouldn't release it.

Note: It is possible for you to create an object by invoking one of the +className... methods of the class object. However, because these class methods allocate memory for the object, they are considered to be the owners and are therefore responsible for releasing the object.

When you write a method that creates and returns an object, that method is responsible for releasing the object. However, it's clearly not fruitful to dispose of an object before the recipient of the object gets it. What is needed is a way to mark an object for release at a later time, so that it will be properly disposed of after the recipient has had a chance to use it. The Foundation Framework provides just such a mechanism.

Marking Objects for Disposal

The **autorelease** method, defined by NSObject, marks the receiver for later release. By autoreleasing an object—that is, by sending it an **autorelease** message—you declare that you don't need the object to exist beyond the scope you sent **autorelease** in. When your code completely finishes executing and control returns to the application object (that is, at the end of the event loop), the application object releases the object. The **sprockets** methods above could be implemented in this way:

```
- (NSArray *)sprockets
{
    NSArray *array;

    array = [[NSArray alloc] initWithObjects:mainSprocket,
                                auxiliarySprocket, nil];
    return [array autorelease];
}
```

When another method gets the array of Sprockets, that method can assume that the array will be disposed of when it's no longer needed, but can still be safely used anywhere within its scope (with certain exceptions; see "Validity of Shared Objects" below). It can even return the array to its invoker, since the application object defines the bottom of the call stack for your code. The **autorelease** method thus allows every object to use other objects without worrying about disposing of them.

Note: Just as it's an error to release an object after it's already been deallocated, it's an error to send so many `autorelease` messages that the object would later be released after it had already been deallocated. You should send `release` or `autorelease` to an object only as many times as are allowed by its creation (one) plus the number of `retain` messages *you* have sent it (`retain` messages are described below).

Retaining Objects

There are times when you don't want a received object to be disposed of; for example, you may need to cache the object in an instance variable. In this case, only you know when the object is no longer needed, so you need the power to ensure that the object is not disposed of while you are still using it. You do this with the **retain** method, which stays the effect of a pending **autorelease** (or preempts a later **release** or **autorelease** message). By retaining an object you ensure that it won't be deallocated until you're done with it. For example, if your object allows its main Sprocket to be set, you might want to retain that Sprocket like this:

```
- (void)setMainSprocket:(Sprocket *)newSprocket
{
    [mainSprocket autorelease];
   mainSprocket = [newSprocket retain]; /* Claim the new Sprocket. */
   return;
}
```

Now, **setMainSprocket:** might get invoked with a Sprocket that the invoker intends to keep around, which means your object would be sharing the Sprocket with that other object. If that object changes the Sprocket, your object's main Sprocket changes. You might want that, but if your Thingamajig needs to have its own Sprocket the method should make a private copy:

```
- (void)setMainSprocket:(Sprocket *)newSprocket
{
    [mainSprocket autorelease];
    mainSprocket = [newSprocket copy]; /* Get a private copy. */
    return;
}
```

Note that both of these methods autorelease the original main sprocket, so they don't need to check that the original main sprocket and the new one are the same. If they simply released the original when it was the same as the new one, that sprocket would be released and possibly deallocated, causing an error as soon as it w as retained or copied. Although they could store the old main sprock et and release it later, that kind of code tends to be slightly more comple x. For example:

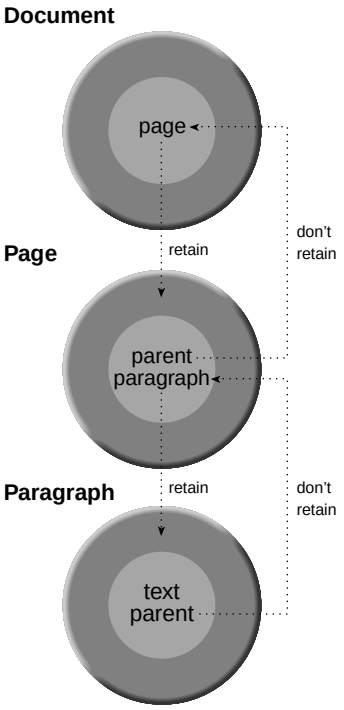
```
- (void)setMainSprocket:(Sprocket *)newSprocket
{
    Sprocket *oldSprocket = mainSprocket;
    mainSprocket = [newSprocket copy];
    [oldSprocket release];
    return;
}
```

Retain Cycles

In general, you retain all objects that you create. However , in some cases you may have two objects with cyclical references; that is, each object contains an instance variable that refers to the other object. For example, consider a text program with the object relationships shown in the following figure. The Document object creates a Page object for each page in the document. Each Page object has an instance variable that keeps track of which document it's in. If the Document object retained the Page object and the Page object retained the Document object, neither object would ever be released. The Document's reference count can't become 0 until the Page object is released, and the Page object won' t be released until the Document object is deallocated.

The solution to the problem of retain c ycles is that the "parent" object should retain its "children," but that the children should not retain their parents. So, in the following figure the document object retains its page objects b ut the page

object does not retain the document object.



Validity of Shared Objects

The Foundation Framework's ownership policy limits itself to the question of when you have to dispose of an object; it doesn't specify that any object received in a method *must* remain valid throughout that method's scope. A received object nearly always becomes invalid when its owner is released, and usually becomes invalid when its owner reassigns the instance variable holding that object. Any method other than **release** that immediately disposes of an object is documented as doing so.

For example, if you ask for an object's main sprocket and then release the object, you have to consider the main sprocket gone, because it belonged to the object. Similarly, if you ask for the main sprocket and then send **setMainSprocket:** you can't assume that the sprocket you received remains valid:

```
Sprocket *oldMainSprocket;
Sprocket *newMainSprocket;

oldMainSprocket = [myObject mainSprocket];

/* If this releases the original Sprocket... */
[myObject setMainSprocket:newMainSprocket];

/* ...then this causes the application to crash. */
[oldMainSprocket anyMessage];
```

setMainSprocket: may release the object's original main sprocket, possibly rendering it invalid. Sending any message to the invalid sprocket would then cause your application to crash. If you need to use an object after disposing of its owner or rendering it invalid by some other means, you can retain and autorelease it before sending the message that would invalidate it:

```
Sprocket *oldMainSprocket;
Sprocket *newMainSprocket;

oldMainSprocket = [[[myObject mainSprocket] retain] autorelease];
[myObject setMainSprocket:newMainSprocket];
[oldMainSprocket anyMessage];
```

Retaining and autoreleasing **oldMainSprocket** guarantees that it will remain valid throughout your scope, even though its owner may release it when you send **setMainSprocket:**.

Summary

Now that the concepts behind the Foundation Framework's object ownership policy have been introduced, they can be expressed as a short list of rules:

- If you directly allocate, copy, or retain an object, you are responsible for releasing the newly created object with `release` or `autorelease`. Any other time you receive an object, you are not responsible for releasing it.
- A received object is normally guaranteed to remain valid within the method it was received in. That method may also safely return the object to its invoker.
- If you need to store a received object in an instance variable, you must retain or copy it.
- Use `retain` and `autorelease` when needed to prevent an object from being invalidated as a normal side-effect of a message.