The iPhone has **no Garbage Collection**. That means that **memory is not automatically freed up** when an object is no longer in use (the object is no longer pointed to by some pointer variable).

The programmer must be careful to free memory when an object is no longer in use.

The methods to do this are **retain, release**, and **dealloc**.

Every object has a block of memory that stores its data. One part of this block is a variable called the **retainCount**. This records the number of pointers with access to the block.

![Diagram showing object memory block with retainCount value=2]

In this example, there are two pointers to the object memory block and its retainCount is 2. The methods retain and release affect the value in retainCount. The method **dealloc is automatically called when the retainCount reaches zero** and dealloc causes the memory to be released. The whole point of using retain and release is to keep the retainCount accurate.

Any method that begins with “**alloc**, “**new**”, or contains the word “**copy**” allocates a memory block for an object and **sets the retainCount to 1**.

The **retain** method **increments the retainCount by 1**.

The **release** method **decrements the retainCount by 1**.

Example:

FrontDoorViewController *myFrontDoor = [[FrontDoorViewController alloc] init];
The block of memory for the object “myFrontDoor” is allocated and the retainCount is set to 1.

The @property directive contains attributes. One of them is frequently retain:

@property (non-atomic, retain) NSArray *tvShows;

The retain attribute tells the compiler to write the setter function to retain any object that is assigned to the instance variable tvShows.

The correct way to allocate memory for tvShows is as follows:

1. NSMutableArray *myArray = [[NSMutableArray alloc] initWithCapacity: 1];
2. self.tvShows = myArray;
3. [myArray release];

Statement 1 allocates a memory block and sets the retainCount to 1. Statement 2 assigns the pointer to the memory block to the instance variable tvShows. Because the setter method for tvShows executes a retain, the retainCount for the block is now 2. Statement 3 sends a release to the memory block, decrementing its retainCount to 1.

When the program is done with tvShows, it will execute a release. This will decrement the retainCount to zero and the memory block will be freed. This typically occurs in the dealloc method for the class:

-(void)dealloc{
    
    [tvShows release];
    
    [super dealloc];
}

It also happens if a new assignment is made to tvShows (see implementation of the setter method below).

The incorrect way to allocate memory for tvShows is as follows:

1. self.tvShows = [[NSMutableArray alloc] init];

The alloc method in Statement 1 allocates a memory block and sets the retainCount to 1. The assignment of the pointer to the memory block to tvShows. Because the setter method executes a retain, the retainCount is now 2. Since there is no release following this
statement, when tvShows is released during the class dealloc, the retainCount will decrement to 1 and the memory block will not be released, even though there is no existing pointer to the memory. This is a **memory leak**.

*****************************************************************************

**Implementation of a setter method**

@property(non-atomic, retain) TreeClass *tree;

**Setter:**

    -(void)setTree: (TreeClass *)newTree {
        1. if (newTree!=tree) {
            2. [tree release];
            3. tree = [newTree retain];
        }
    }

Statement 2 releases the current value of tree. This the release on reassignment mentioned above. This will decrement the retainCount for the memory block pointed to by tree and if it goes to zero, the memory block will be deallocated.
Statement 3 executes a retain on the memory block pointed to by newTree as stated above before assignment to tree.
Statement 1 checks that the pointers in newTree and tree are not the same. In case they are the same, we don’t want to release the item before we do the assignment. The retainCount could go to zero and the pointer would be meaningless and cause an error.

*****************************************************************************

**Implementation of a getter method**

**Getter:**

    -(TreeClass *) tree {
        return(tree)
    }

*****************************************************************************

**autorelease**

Autorelease is an alternative to release. Instead of immediately decreasing the retainCount by 1, autorelease marks a memory block for later retainCount decrease by adding the block
to an autorelease pool. At some later point, everything in the pool has its retainCount decreased by 1. If at that time, the retainCount becomes zero, the memory block is freed.

Note that in the above creation of an NS MUTABLE ARRAY tvShows, we used alloc to create the array and so had to release it after assigning the pointer to tvShows. An alternate way to allocate memory for tvShows is the following which doesn’t use a method with “alloc”, “new”, or “copy”.

1. self.tvShows = [NSMutableArray arrayWithCapacity: 1];

The assignment to tvShows executes a retain as before and it must be released in the dealloc method as before. But, we don’t do any other release here. How is this possible, since the creation of the NS MUTABLE ARRAY must cause the retainCount to become 1?

The answer is that the method arrayWithCapacity uses autorelease to be sure the retainCount will eventually be decreased to zero:

+(NSMutableArray *)arrayWithCapacity:(NSUInteger)numItems{
    NSMutableArray *newArray = [[alloc] ...];
    [newArray autorelease];
    return(newArray);
}

In general, use autorelease when you create an object with “alloc”, “new”, or “copy”, but can’t immediately release it.

Avoiding cyclic retain

Two related objects must not both retain each other, or neither can be freed by the dealloc method. This is common, for example, when a UIViewController has an instance variable which is another UIViewController for a subview.

@interface TopViewController: UIViewController{
    ChildViewController *childController;
}

@property (nonatomic, retain) ChildViewController *childController;

@interface ChildViewController: UIViewController{
    TopViewController *myParentViewController;
}

@property (nonatomic) TopViewController *myParentViewController;
}
Notice that the child does not retain the parent. If it did, both memory blocks would have a retain count of at least one and neither could go to zero.

********************************************************************************
Why to use self.myObject when doing assignment.

Because the dot notation is a shortcut for the setter method, using

    self.myObject = createdObject

does a retain on the createdObject memory block which is correct.

If instead, we used

    myObject = createdObject

Then the assignment is direct, without using the setter method and no retain is performed. The memory block will disappear as soon as a release method is called.