# Objective C Properties

* Property is basically an instance variable. It has the following functions:
  + 1. Declares an instance variable with a different name (images -> \_images) to hide it.
  + 2. Creates a getter with the same name
  + 3. Creates a setter (images -> setImages)
  + 4. Allows to use the dot notation
* The following defaults apply to the first declaration:
  + 1. For memory management a strong(default), weak, copy, assign (retain is a synonym of strong).
  + 2. For thread safety atomic(default), nonatomic
  + 3. For mutability readwrite(default), readonly
* The attributes of property are as follows:
  + **atomic -** This is by default. When atomic is used then the synthesize method uses a lock to serialize read/write to this variable. However, this is not fast enough
  + **nonatomic -** This is to avoid locking so does not have multithreading properties but is faster than atomic.
  + **strong -** This is by default. If a variable is strong the ARC will increase the counter. So ARC keeps the object that is pointed in the memory until that class instance stops. This is usually what we want but can be problems with retail cycle
  + **weak -** In this case, ARC does not increment the retain count. It keeps the pointer until another class points to it strongly. An example where this would be useful is if you had two classes, one for a petOwner, and one for their pet. Lets say for some reason we want them to refer to each other, so you can request the pet of an owner, or an owner of a pet. If the pet’s petOwner property, and the petOwner’s pet property were both strong, the memory could never be released because they would both be telling ARC that they both need it. If we set the petOwner’s pet property to weak, then we avoid the retain cycle. If the pet object is destroyed, while the petOwner still has a reference to it, the petOwner’s pet property will be automatically set to nil.
  + **assign -** A pre-ARC attribute. This is the default types for primitive types like int, long or float. Similar to weak. This basically says no memory management, just assign.
  + **copy -** The recommendation is to use copy for classes that are part of a class cluster that have mutable/immutable pairs; such as NSString/NSMutableString NSArray/NSMutableArray NSDictionary/NSMutableDictionary NSSet/NSMutableSet. The reason for this is that it is possible to have a property that is declared as an immutable type (such as NSString) yet pass it a mutable type (such as NSMutableString). In which case it is possible to change the property from outside the class as Taskinoor describes. Using copy is recommended, because it behaves sensibly with class clusters. sending copy to a mutable class returns an immutable copy of the object (i.e. sending a copy message to an NSMutableString returns an NSString). But, sending copy to an immutable counterpart is equivalent to sending it a retain message.
  + As an example, the class interface for the XYZBadgeView class shown earlier in [Figure 3-4](https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/EncapsulatingData/EncapsulatingData.html#//apple_ref/doc/uid/TP40011210-CH5-SW19) might look like this:

|  |
| --- |
| * + @interface XYZBadgeView : NSView |
| * + @property NSString \*firstName; |
| * + @property NSString \*lastName; |
| * + @end |

* + Two NSString properties are declared, which both maintain implicit strong references to their objects.
  + Consider what happens if another object creates a string to set as one of the badge view’s properties, like this:

|  |
| --- |
| * + NSMutableString \*nameString = [NSMutableString stringWithString:@"John"]; |
| * + self.badgeView.firstName = nameString; |

* + This is perfectly valid, because NSMutableString is a subclass of NSString. Although the badge view thinks it’s dealing with an NSString instance, it’s actually dealing with an NSMutableString.
  + This means that the string can change:

|  |
| --- |
| * + [nameString appendString:@"ny"]; |

* + In this case, although the name was “John” at the time it was originally set for the badge view’s firstName property, it’s now “Johnny” because the mutable string was changed.
  + You might choose that the badge view should maintain its own copies of any strings set for its firstName and lastName properties, so that it effectively captures the strings at the time that the properties are set. By adding a copy attribute to the two property declarations:

|  |
| --- |
| * + @interface XYZBadgeView : NSView |
| * + @property (copy) NSString \*firstName; |
| * + @property (copy) NSString \*lastName; |
| * + @end |

* + **retain -** A pre-ARC attribute. This is the older version of strong. It claims ownership of the object and increases the retain count. You will have to manually release the object to decrease its retain count (which will release it from memory when the retain count goes to 0). You should not use this in an ARC project.
  + **readwrite -** This is the default. This tells the compiler to set up both getter and setter.
  + **read -** This only sets up the getter. One common thing to do with these two, is if you want a property visible to another class, but not able to be changed by an external class, set the property to readonly in your header file, and in your implementation file ( .m file), declare it as readwrite. That way your implementation has a getter and a setter, while external classes only get the getter.
* You can also set a getter and setter of your name using getter="" and setter="".
* General rules are as you say.
  + 1. Object pointers should usually be strong.
  + 2. Primitive types should be assign.
  + 3. weak should be used in child/parent references to avoid reference cycles. Typically the parent has a strong reference to its children and the children have a weak reference to their parent. Delegates are typically weak for the same reason.
  + 4. copy is typically used for NSString, NSArray, NSDictionary, etc. to avoid issues when they are assigned the mutable variant. This avoids the problem of the value being changed unexpectedly.
  + 5. There's a big "gotcha" using copy with NSMutableString, NSMutableArray, etc. because when you assign the mutable value to the property, the copy attribute results in the copy method being called which gives back a non-mutable copy of the original value. The solution is to override the setter method to call mutableCopy.
* Using the wrong attribute could have serious problems depending on the needs of the property and the attribute being used.
  + 1. Using assign instead of strong for an object pointer is probably the worst mistake. It can lead to app crashes due to trying to access deallocated objects.
  + 2. Using nonatomic instead of atomic on a property that will be accessed concurrently on multiple threads may lead to really hard to find bugs and/or crashes.
  + 3. Using strong instead of copy for NSString or NSArray (and other collections) can possibly lead to subtle and hard to find bugs if mutable variants were assigned to the property and other code later modifies those values.