Variable property attributes or Modifiers in iOS

Variable property attributes or Modifiers

Property Attributes Indicate Data Accessibility and Storage Considerations

Use Accessor Methods to Get or Set Property Values

01. atomic //default
02. nonatomic
03. strong=retain //default
04. weak= unsafe_unretained
05. retain
06. assign //default
07. unsafe_unretained
08. copy
09. readonly
10. readwrite //default

01. atomic

-Atomic means only one thread access the variable(static type).

-Atomic is thread safe.

-but it is slow in performance

-atomic is default behavior

-Atomic accessors in a non garbage collected environment (i.e. when using retain/release/autorelease) will use a lock to

ensure that another thread doesn't interfere with the correct setting/getting of the value.

-it is not actually a keyword.

Example :

```
@property (retain) NSString *name;
```

```
@synthesize name;
```

02. nonatomic

-Nonatomic means multiple thread access the variable(dynamic type).

-Nonatomic is thread unsafe.

-but it is fast in performance

-Nonatomic is NOT default behavior,we need to add nonatomic keyword in property attribute.

-it may result in unexpected behavior, when two different process (threads) access the same variable at the same time.

Example:

```
@property (nonatomic, retain) NSString *name;
```

@synthesize name;

Explain:

Suppose there is an atomic string property called "name", and if you call [self setName:@"A"] from thread A,

call [self setName:@"B"] from thread B, and call [self name] from thread C, then all operation on different thread will be performed serially which means if one thread is executing setter or getter, then other threads will wait. This makes property "name" read/write safe but if another thread D calls [name release] simultaneously then this operation might produce a crash because there is no setter/getter call involved here. Which means an object is read/write safe (ATOMIC) but not thread safe as another threads can simultaneously send any type of messages to the object. Developer should ensure thread safety for such objects.

If the property "name" was nonatomic, then all threads in above example - A,B, C and D will execute simultaneously producing any unpredictable result. In case of atomic, Either one of A, B or C will execute first but D can still execute in parallel.

03. strong (iOS4 = retain)

-it says "keep this in the heap until I don't point to it anymore"

-in other words " I'am the owner, you cannot dealloc this before aim fine with that same as retain"

-You use strong only if you need to retain the object.

-By default all instance variables and local variables are strong pointers.

-We generally use strong for UIViewControllers (UI item's parents)

-strong is used with ARC and it basically helps you , by not having to worry about the retain count of an object. ARC automatically releases it for you when you are done with it.Using the keyword strong means that you own the object.

Example:

```
@property (strong, nonatomic) ViewController *viewController;

@synthesize viewController;
```

04. weak (iOS4 = unsafe_unretained)

-it says "keep this as long as someone else points to it strongly"

-the same thing as assign, no retain or release

-A "weak" reference is a reference that you do not retain.

-We generally use weak for IBOutlets (UIViewController's Childs).This works because the child object only

needs to exist as long as the parent object does.

-a weak reference is a reference that does not protect the referenced object from collection by a garbage collector.

-Weak is essentially assign, a unretained property. Except the when the object is deallocated the weak pointer is automatically set to nil

Example :

```
@property (weak, nonatomic) IBOutlet UIButton *myButton;

@synthesize myButton;
```

Explain:

Imagine our object is a dog, and that the dog wants to run away (be deallocated).

Strong pointers are like a leash on the dog. As long as you have the leash attached to the dog, the dog will not run away. If five people attach their leash to one dog, (five strong pointers to one object), then the dog will not run away until all five leashes are detached.

Weak pointers, on the other hand, are like little kids pointing at the dog and saying "Look! A dog!" As long as the dog is still on the leash, the little kids can still see the dog, and they'll still point to it. As soon as all the leashes are detached, though, the dog runs away no matter how many little kids are pointing to it.

As soon as the last strong pointer (leash) no longer points to an object, the object will be deallocated, and all weak pointers will be zeroed out.

When we use weak?

The only time you would want to use weak, is if you wanted to avoid retain cycles

(e.g. the parent retains the child and the child retains the parent so neither is ever released).

05. retain = strong

-it is retained, old value is released and it is assigned

-retain specifies the new value should be sent -retain on assignment and the old value sent -release

-retain is the same as strong.

-apple says if you write retain it will auto converted/work like strong only.

-methods like "alloc" include an implicit "retain"

Example:

```
@property (nonatomic, retain) NSString *name;
```

```
@synthesize name;
```

06. assign

-assign is the default and simply performs a variable assignment

-assign is a property attribute that tells the compiler how to synthesize the property's setter implementation

-I would use assign for C primitive properties and weak for weak references to Objective-C objects.

Example:

```
@property (nonatomic, assign) NSString *address;

@synthesize address;
```

07. unsafe_unretained

-unsafe_unretained is an ownership qualifier that tells ARC how to insert retain/release calls

-unsafe_unretained is the ARC version of assign.

Example:

```
@property (nonatomic, unsafe_unretained) NSString *nickName;

@synthesize nickName;
```

08. copy

-copy is required when the object is mutable.

-copy specifies the new value should be sent -copy on assignment and the old value sent -release.

-copy is like retain returns an object which you must explicitly release (e.g., in dealloc) in non-garbage collected environments.

-if you use copy then you still need to release that in dealloc.

-Use this if you need the value of the object as it is at this moment, and you don't want that value to reflect any changes made by other

owners of the object. You will need to release the object when you are finished with it because you are retaining the copy.

Example:

```
@property (nonatomic, copy) NSArray *myArray;

@synthesize myArray;
```

09. readonly

-declaring your property as readonly you tell compiler to not generate setter method automatically.

-Indicates that the property is read-only.

-If you specify readonly, only a getter method is required in the @implementation block. If you use the @synthesize directive in

the @implementation block, only the getter method is synthesized. Moreover, if you attempt to assign a value using the dot syntax,

you get a compiler error.

Example:

```
@property (nonatomic, readonly) NSString *name;

@synthesize name;
```

10. readwrite

-setter and getter generated.

-Indicates that the property should be treated as read/write.

-This attribute is the default.

-Both a getter and setter method are required in the @implementation block. If you use the @synthesize directive in the implementation

block, the getter and setter methods are synthesized.

Example:

```
@property (nonatomic, readwrite) NSString *name;

@synthesize name;
```