`@property ( Memory Management Policies ) returntype propertyName;`

During lecture today we discussed the use of property modifiers such as `nonatomic` and `assign`. These modifiers enable you to set limits in memory that are enforced by the ObjC compiler at runtime, and increase code stability in the process.

As such, these can be seen as per-property memory management policies or configuration profiles that help to restrict object Here is a summary table listing each and how it's used.

`nonatomic` – NOT thread-safe. The default is atomic, meaning the synthesized accessors or getters and setters for the property, are automatically locked into memory once created, which ensures greater stability at runtime in multithreaded applications.

By setting this property as nonatomic you are saying that for better performance you're not particularly bothered by its memory locations being locked (and prevented from changing).

`strong / retain` – Default. Functionally equivalent. The instance variable is a strong reference : When a value is assigned to the instance variable ARC (the ObjC runtime using Automated Reference Counting) will hold on (`retain`) to this new value, and `release` its existing value.

`copy` – The same effect as strong and retain, except the setter copies the incoming value (by sending `copy` to it) and the copy, which has an increased retain count already, **is assigned to the instance variable**. This is used a lot under the hood in Apple's own NSArray and NSMutableArray, and NSString / NSMutableString , where the immutable parent class has a mutable subclass: By making an immutable copy of an instance variable getter, the compiler prevents that getter from mutating the value of that variable, and causing a crash.

`weak` – Weak references assign incoming values to instance variables, but do not retain them after their usage (ie within a method's scope). ARC then sees there are no pointer references existing, and thus turns it to nil and destroys it.

`assign` – This is a pre-ARC (ie manual reference counting, a total pain in the ass long ago) policy, and functions much the same as weak. The setter doesn't manage memory, and the incoming value is assigned directly to the instance variable. The ivar is NOT a weak reference here, however, and will not be destroyed or turned to nil automatically if its instance ceases to exist. This is not memory-safe and can lead to 'dangling pointers'