How iOS handles memory management with ARC

1) How does it work, a break down

Swift uses *Automatic Reference Counting* (ARC) to track and manage your app's memory usage. In most cases, this means that memory management "just works" in Swift, and you don't need to think about memory management yourself. ARC automatically frees up the memory used by class instances when those instances are no longer needed.

However, in a few cases ARC requires more information about the relationships between parts of your code in order to manage memory for you. This chapter describes those situations and shows how you enable ARC to manage all of your app's memory. Using ARC in Swift is very similar to the approach described in Transitioning to ARC Release Notes for using ARC with Objective-C.

Reference counting applies only to instances of classes. Structures and enumerations are value types, not reference types, and aren't stored and passed by reference.

How ARC Works
Every time you create a new instance of a class, ARC allocates a chunk of memory to store information about that instance. This memory holds information about the type of the instance, together with the values of any stored properties associated with that instance.

Additionally, when an instance is no longer needed, ARC frees up the memory used by that instance so that the memory can be used for other purposes instead. This ensures that class instances don't take up space in memory when they're no longer needed.

However, if ARC were to deallocate an instance that was still in use, it would no longer be possible to access that instance's properties, or call that instance's methods. Indeed, if you tried to access the instance, your app would most likely crash.

To make sure that instances don't disappear while they're still needed, ARC tracks how many properties, constants, and variables are currently referring to each class instance. ARC will not deallocate an instance as long as at least one active reference to that instance still exists.

To make this possible, whenever you assign a class instance to a property, constant, or variable, that property, constant, or variable makes a *strong reference* to the instance. The reference is called a “strong” reference because it keeps a firm hold on that instance, and doesn’t allow it to be deallocated for as long as that strong reference remains.

2) Compare it to how we did it before ARC was created (MRR / MRC)

In ARC you don't have to release/autorelease the memory allocated by you where as in case of manual you have to take care of this. e.g. manual case

```
-(void)someMethod
{
   NSMutableArray *arr = [[NSMutableArray alloc] init];
   //use array
   [arr release]; //when array is in no use
}
```

ARC case

```
-(void)someMethod
{
   NSMutableArray *arr = [[NSMutableArray alloc] init];
   //use array
}
```

3) Compare and contrast ARC to a garbage collector like in Java

Garbage Collection

Garbage Collection (or GC for short) is the technique used for life cycle management on the .NET and Java platforms. The way GC works is that the runtime (either the Common Language Runtime for .NET or the Java Runtime) has infrastructure in place that detects unused objects and object graphs in the background.

This happens at indeterminate intervals (either after a certain amount of time has passed, or when the runtime sees available memory getting low), so objects are not necessarily released at the *exact moment* they are no longer used.

Advantages of Garbage Collection

- GC can clean up entire object graphs, including retain cycles.
- GC happens in the background, so less memory management work is done as part of the regular application flow.

Disadvantages of Garbage Collection

- Because GC happens in the background, the exact time frame for object releases is undetermined.
- When a GC happens, other threads in the application may be temporarily put on hold.

Automatic Reference Counting

Automatic Reference Counting (ARC for short) as used on Cocoa takes a different approach. Rather than having the runtime look for and dispose of unused objects in the background, the compiler will inject code into the executable that keeps track of object reference counts and will release objects as necessary, automatically. In essence, if you were to disassemble an executable compiled with ARC, it would look (conceptually) as if the developer spent a lot of time meticulously keeping track of object life cycles when writing the code — except that all that hard work was done by the compiler.

Advantages of Automatic Reference Counting

- Real-time, deterministic destruction of objects as they become unused.
- No background processing, which makes it more efficient on lower-power systems, such as mobile devices.

Disadvantages of Automatic Reference Counting

- Cannot cope with retain cycles.

Retain Cycles

A so-called retain cycle happens when two (or more) objects reference each other, essentially keeping each other alive even after all external references to the objects have gone out of scope. The Garbage Collection works by looking at "reachable" objects, it can handle retain cycles fine, and will discard entire object graphs that reference each other, if it detects no outside references exist.

Because Automatic Reference Counting works on a lower level and manages life cycles based on reference counts, it cannot handle retain cycles automatically, and a retain cycle will cause objects to stay in memory, essentially causing the application to "leak" memory.

ARC provides a method to avoid retain cycles, but it does require some explicit thought and design by the developer. To achieve this, ARC introduces Storage Modifiers that can be applied to object references (such as fields or properties) to specify how the reference will behave. By default, references are strong, which means that they will behave as described above, and storing an object reference will force the object to stay alive until the reference is removed. Alternatively, a reference can be marked as weak. In this case, the reference will not keep the object alive, instead, if all *other* references to the stored object go away, the object will indeed be freed and the reference will automatically be set to nil.

A common scenario is to determine a well-defined parent/child or owner/owned relationship between two objects that would otherwise introduce a retain cycle. The parent/owner will maintain a regular reference to the child, while the child or owned object will merely get a weak reference to the parent. This way, the parent can control the (minimum) lifetime of the child, but when the parent object can be freed, the references from the children won't keep it alive.

Of course, the children or owned objects need to be implemented in a way that enables them to cope with the parent reference going nil (which would, for example, happen if an external direct reference to the child kept it alive, while the parent is destroyed). It would be up to the developer to determine how to handle such a scenario, depending on whether the child object is able to function without the parent or not.

The Storage Modifiers are only supported on Cocoa.

References

https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html

https://stackoverflow.com/questions/9110188/difference-between-arc-and-mrc

https://docs.elementscompiler.com/Concepts/ARCvsGC/