iOS Memory Management - Part II

Q) What happens behind the scenes when ARC deallocates an object?

A) ARC automatically deallocates objects when their reference count reaches zero. It releases the memory and removes any associated resources.

Q) How does ARC work with value types (structs, enums) vs reference types (classes)?

A) ARC only applies to reference types (classes). Value types (structs, enums) are stored on the stack and copied when assigned to new variables.

Q) What are common causes of memory leaks in iOS apps?

A) - Retain cycles (strong references between objects)
- Unreleased objects in collections (e.g., arrays, dictionaries)
- Capturing self strongly in closures
- Retaining objects too long in caches

Q) How can we debug memory leaks in iOS?

A) - Use **Xcode Instruments (Leaks, Allocations, Zombies)**
- Analyze memory using **Memory Graph Debugger**
- Enable **Malloc Scribble & Zombies** to detect use-after-free bugs

Q) What is the difference between escaping and non-escaping closures?

A) - **Escaping closures** outlive their scope and are stored for later execution.
- **Non-escaping closures** are executed within the function call and cannot be retained.

Q) How can closures cause retain cycles?

A) Closures capture `self` strongly by default, preventing ARC from deallocating the object, leading to retain cycles.

Q) How to prevent retain cycles in closures?

A) Use `[weak self]` or `[unowned self]` to break the strong reference inside closures.

Example:

```
myClosure = { [weak self] in
    self?.doSomething()
}
```

Q) What are retain cycles in delegation patterns, and how can they be avoided?

A) A delegate property is usually defined as `weak` to prevent a retain cycle between the delegate and the delegating object.

Example:

```
protocol MyDelegate: AnyObject {
    func doSomething()
}

class MyClass {
    weak var delegate: MyDelegate?
}
```

Q) How does NotificationCenter cause memory leaks, and how to avoid it?

A) Observers registered with `NotificationCenter` must be explicitly removed, or they may cause memory leaks.

Solution:

```
NotificationCenter.default.removeObserver(self)
```

Q) How does memory management work with GCD (Grand Central Dispatch)?

A) - Strong references in **async** blocks can lead to retain cycles.
   - Use `[weak self]` inside GCD blocks to avoid capturing `self` strongly.

Example:

```
DispatchQueue.global().async { [weak self] in
    self?.doBackgroundTask()
}
```

Q) How does async/await affect memory management?

A) **Async/await** captures variables like closures, so using `weak self` is recommended when working with long-running async tasks.

Q) What are race conditions & memory leaks in multi-threading?

A) - Race conditions occur when multiple threads access shared memory without synchronization.

- Memory leaks happen when objects are retained due to improper concurrency handling.