ARC &
Using Open Source Packages

4/14’ 16

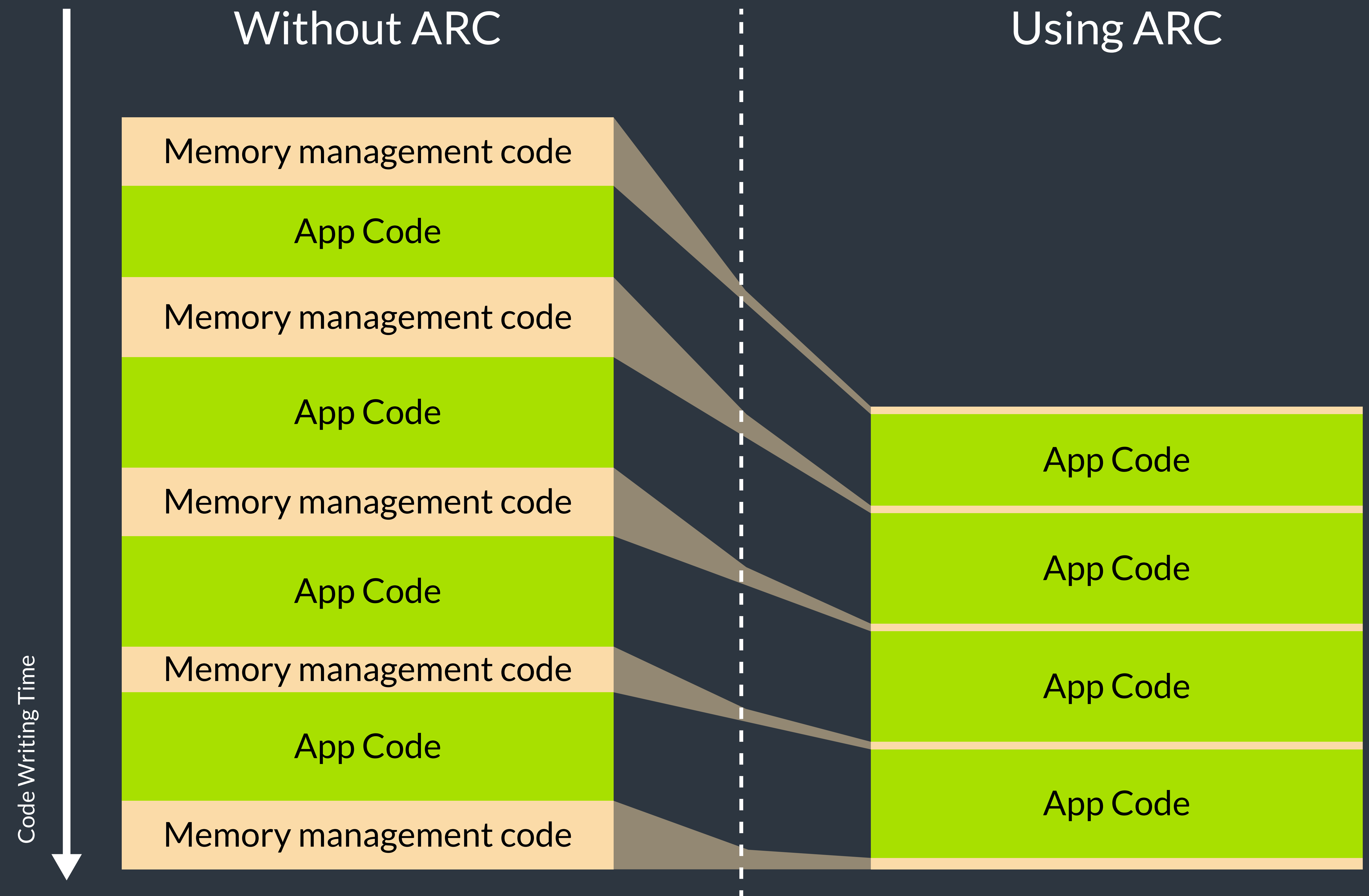
Memory Management in Swift

Memory Management in Swift

- We usually use "Reference counting" and a reference graph to manage dynamically allocated objects in memory usage.
- The Swift language uses ARC, *Automatic Reference Counting*, for memory management.
  This is "Reference Counting". So it's only related to "class" types.
- Most modern languages use GC, Garbage Collection, which requires a background process and slows down apps.

Memory Management in Swift

- Before ARC (iOS 5), Apple's platforms use MRC, *Manual Reference Counting*.



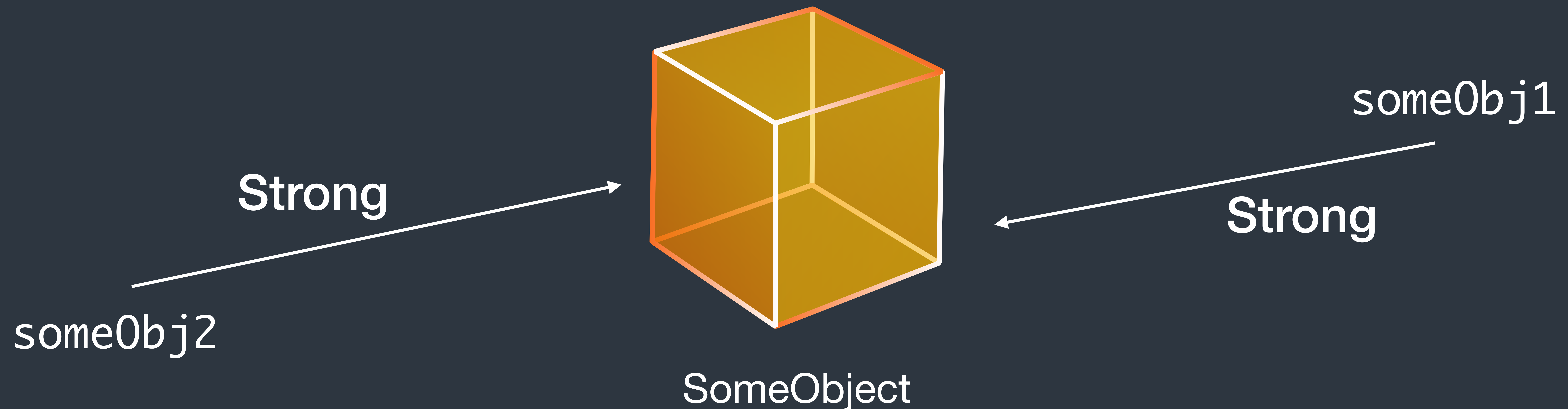
Memory Management in Swift

- ARC lifts the work of tracking the reference graph of all objects from the programmers to the compiler.
  The whole memory management thing gets fully processed during compile time. The performance of runtime is not affected.
- Objects are freed by ARC as soon as when they're not pointed by any references.
- But you have to tell ARC some information of your object graph when necessary.

Memory Management in Swift

- “References to objects” as “Ownership of objects”
- 3 primary reference types: Strong, Weak, and Unowned
- Strong reference keep object alive in memory.
  Objects would be destroyed when there’s no strong refs pointing to them.
- Weak and Unowned references just point to the object but not keep objects alive.
  When the object is deallocated, weak references will point to “nil” instead.

The Strong Reference



- A strong reference keeps object alive in memory.
  An object would be destroyed when there's no any strong references pointing to it.

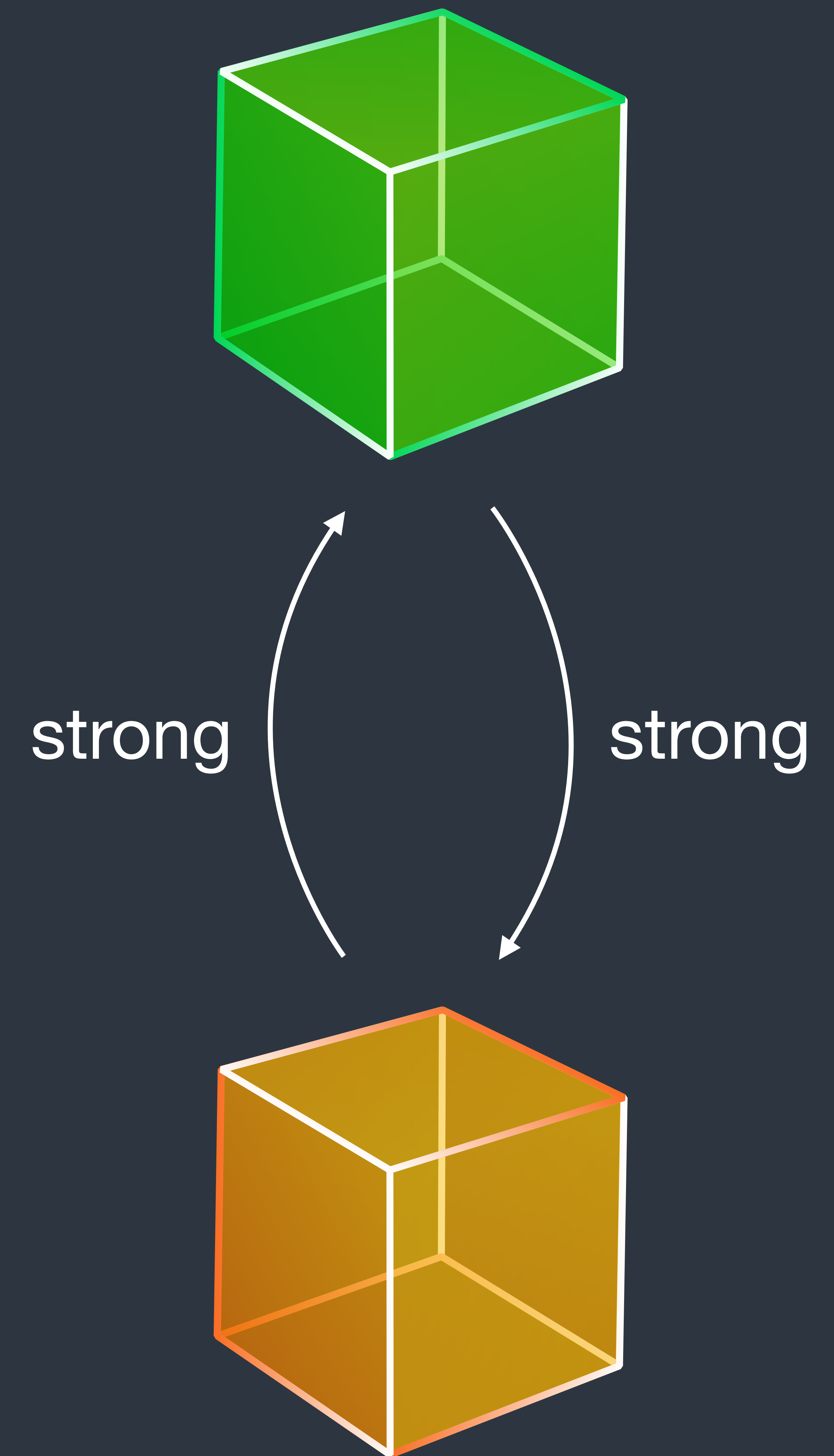
The Strong Reference

SomeObject

- A strong reference keeps object alive in memory.
  An object would be destroyed when there’s no any strong references pointing to it.

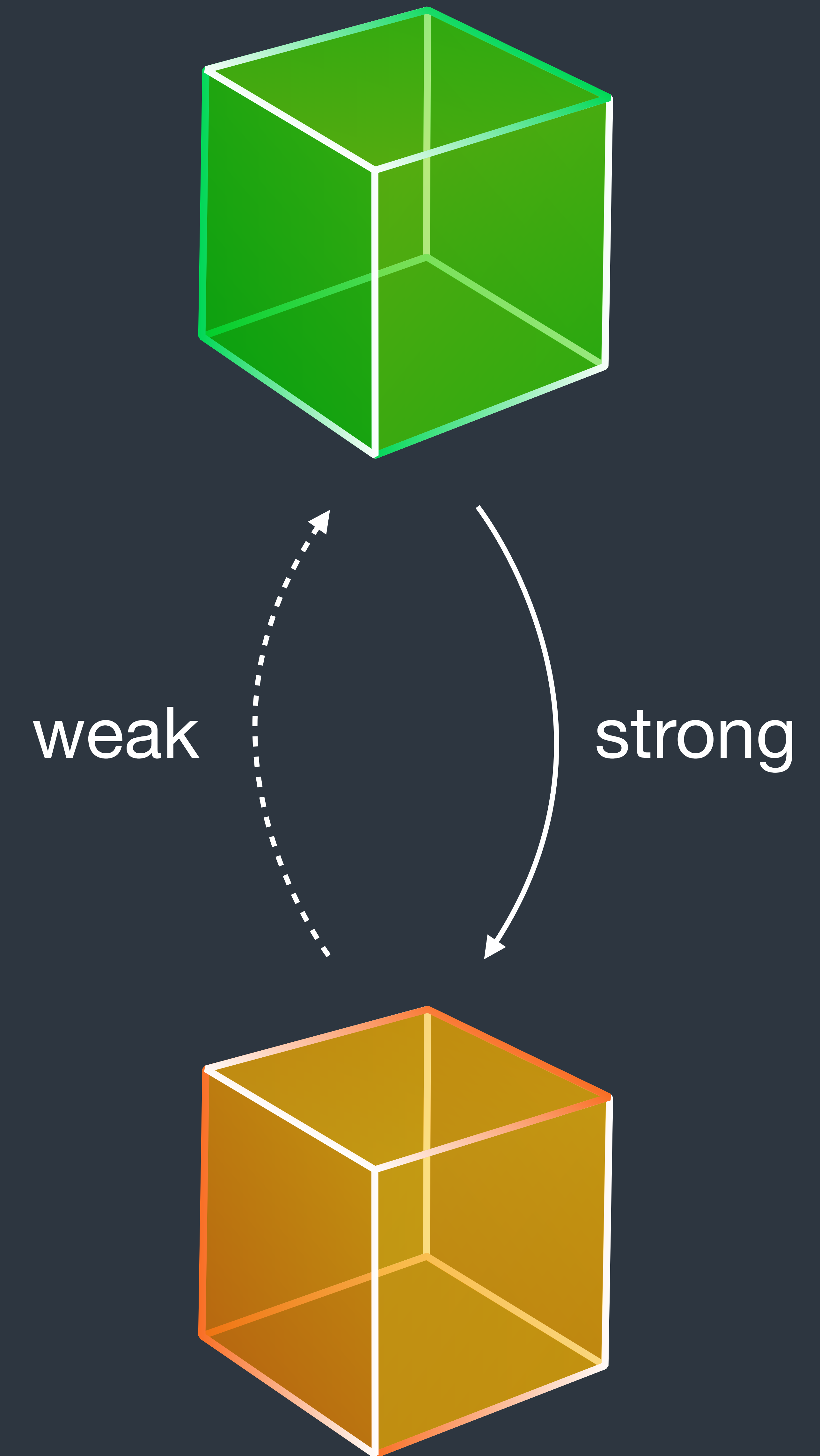
Strong Reference Cycle

- A strong reference cycle keeps all related objects living in the memory even when no references are pointing to those objects.
- Memory resources of mobile devices are limited. Having lots of un-recycled resources would make your app crashing.

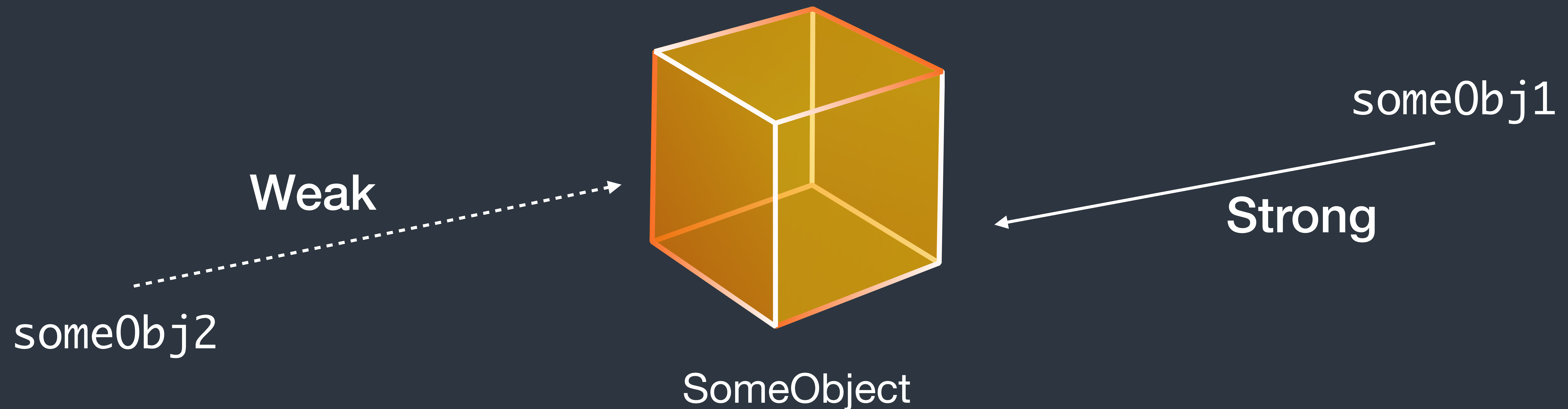


Strong Reference Cycle

- Weak and unowned references are designed to solve the strong reference cycle issue.

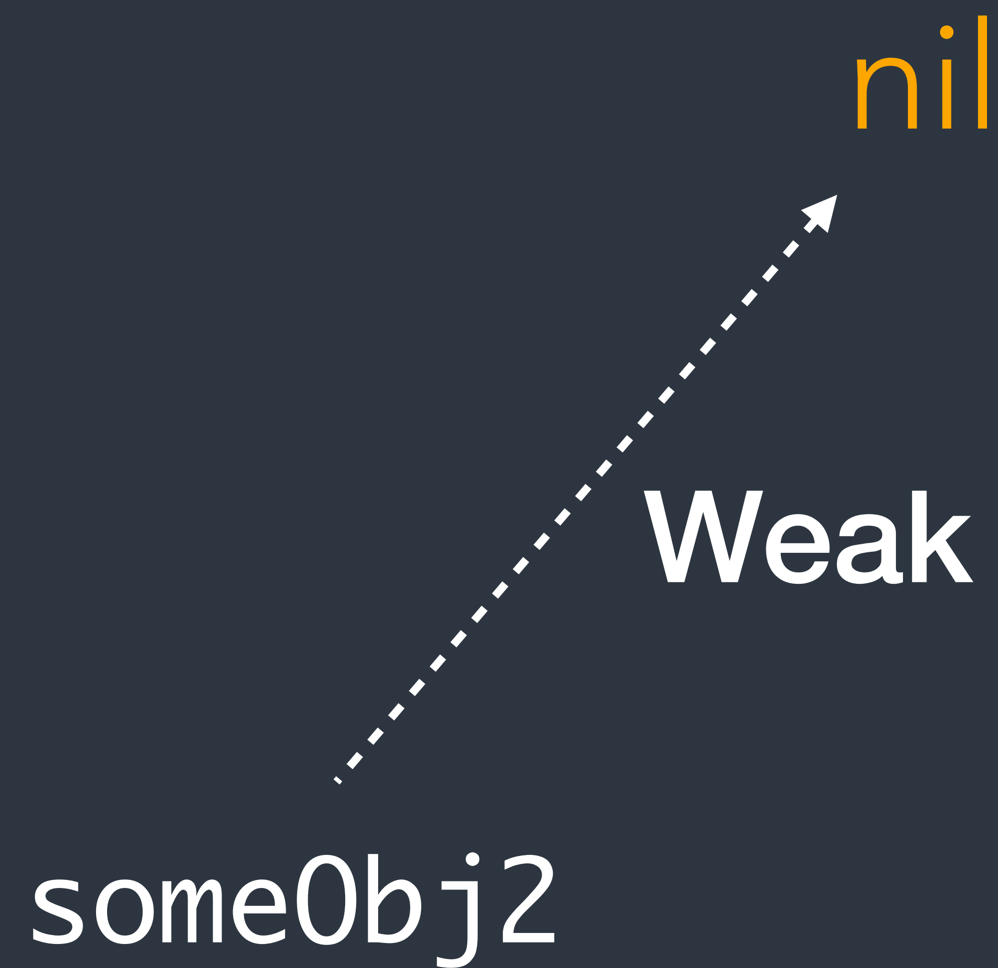


The Weak Reference



- Weak reference just points to the object. It doesn't keep object alive.
  When the object is deallocated, the reference will point to "nil" instead.

The Weak Reference



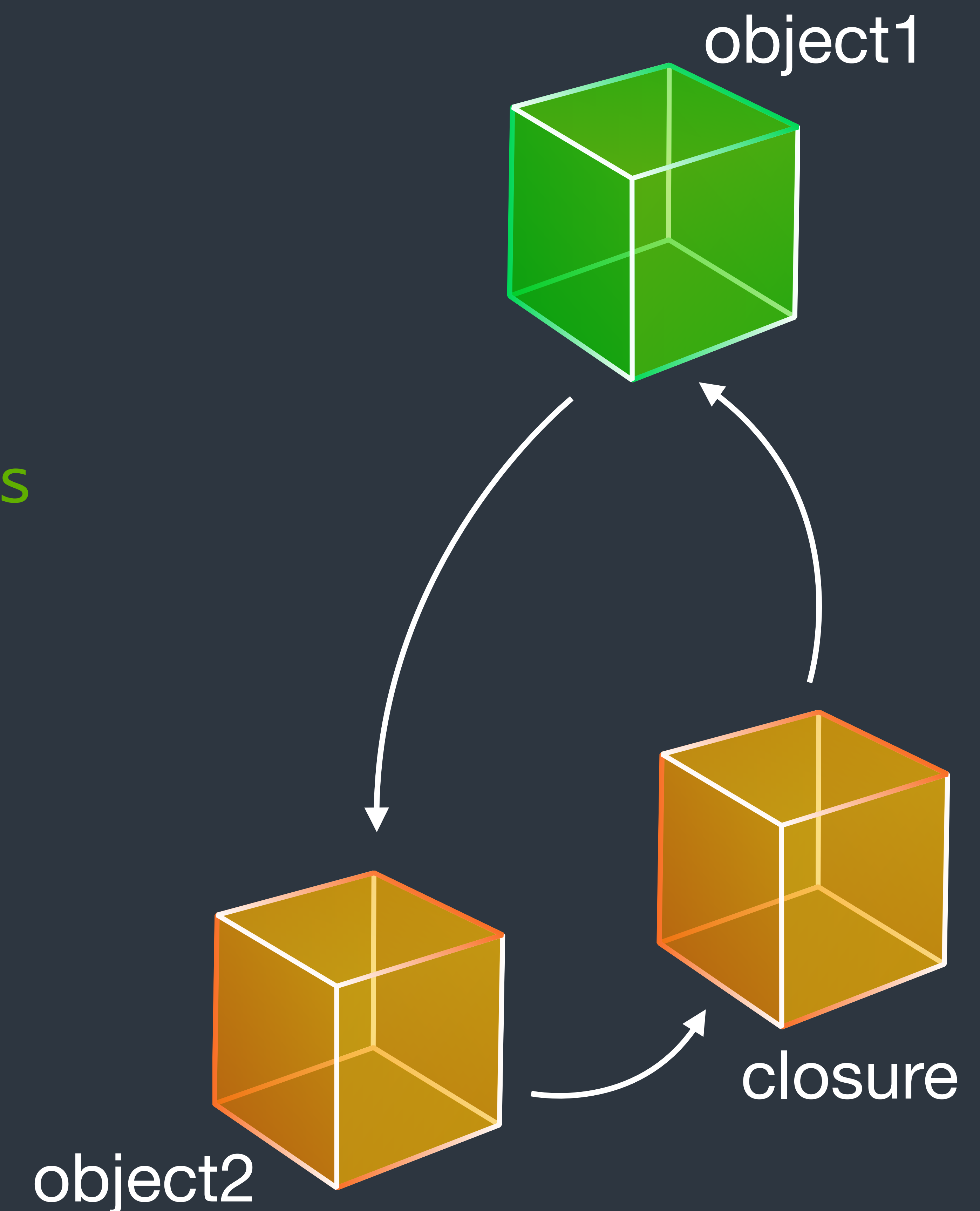
- Weak reference just points to the object. It doesn't keep object alive.
  When the object is deallocated, the reference will point to "nil" instead.

Weak and Unowned References

- Weak references would point to “nil” after its target object is freed. And hence weak references must be Optionals.
- Conversely, use an unowned reference when you know that the reference will never be nil once it has been set during initialization.

Strong Reference Cycle

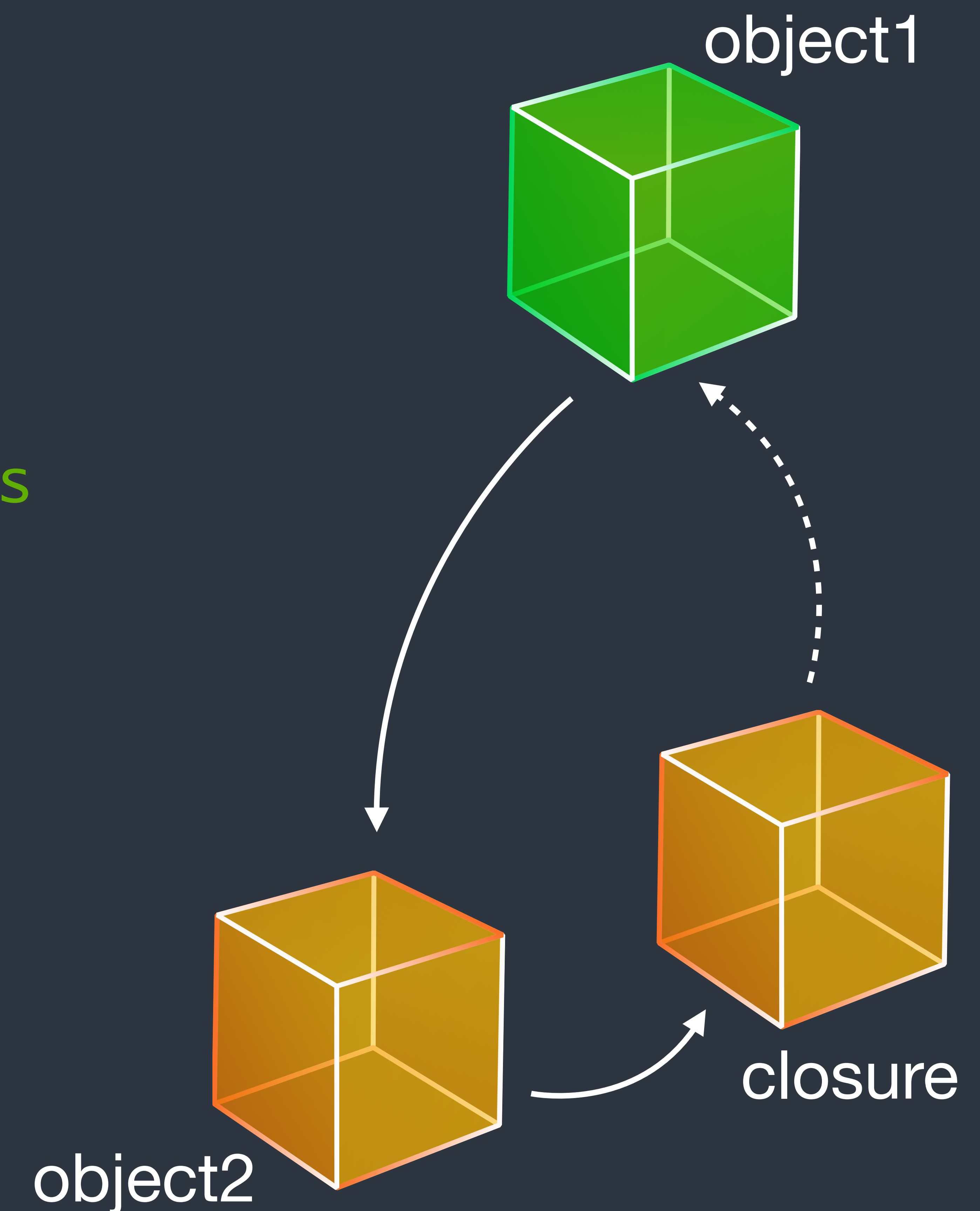
```
let object1 = SomeObject()  // Assume SomeObject is a class
let object2 = AnotherObject()  // Assume AnotherObject is a class

object1.anotherStrongRefObject = object2

object2.strongRefClosure = {
    // Closures would have refs to objects they used.
    print(object1)
}
```



- Closures hold used variables via strong references by default. It’s your responsibility to deal with the reference cycle issue.

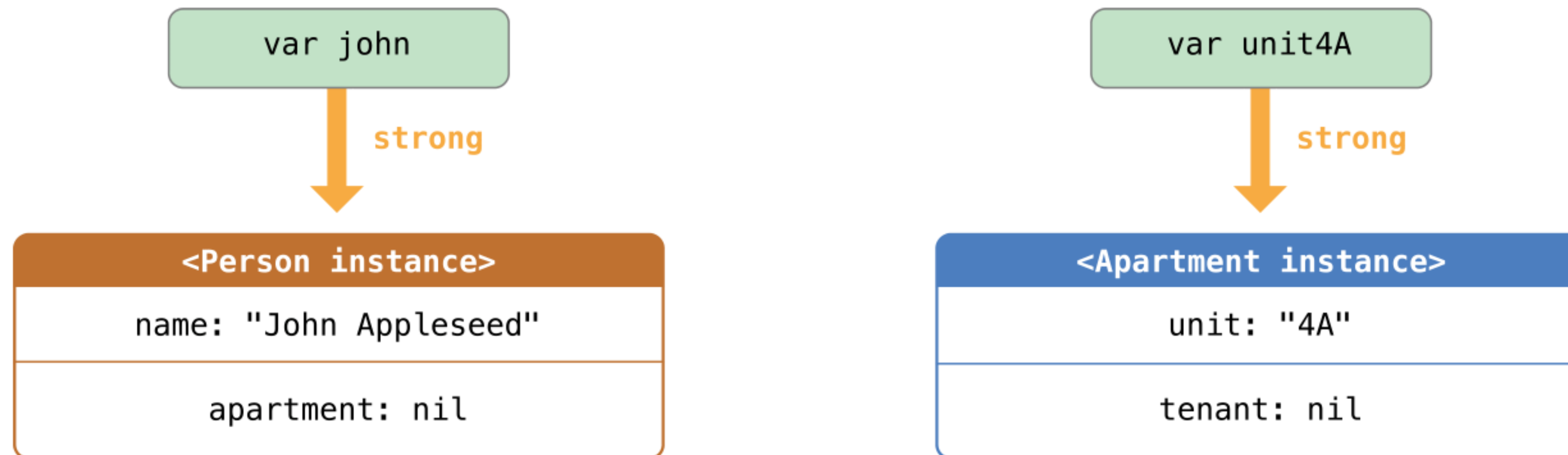
Strong Reference Cycle

```
let object1 = SomeObject()  // Assume SomeObject is a class
let object2 = AnotherObject()  // Assume AnotherObject is a class

object1.anotherStrongRefObject = object2

object2.strongRefClosure = { [unowned object1] in
    // Closures would have refs to objects they used.
    print(object1)
}
```

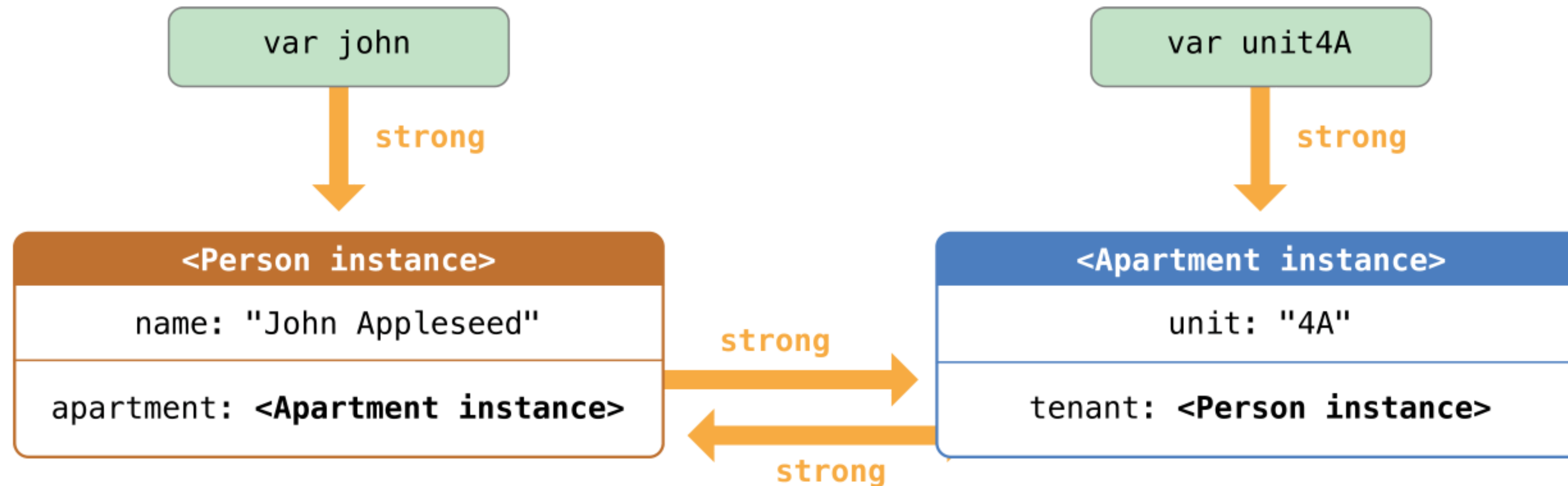


- Use a "Capture List" to define how closures capture values.
  `[ref-type1 var1, ref-type2 expr2, ref-type3 var3]`

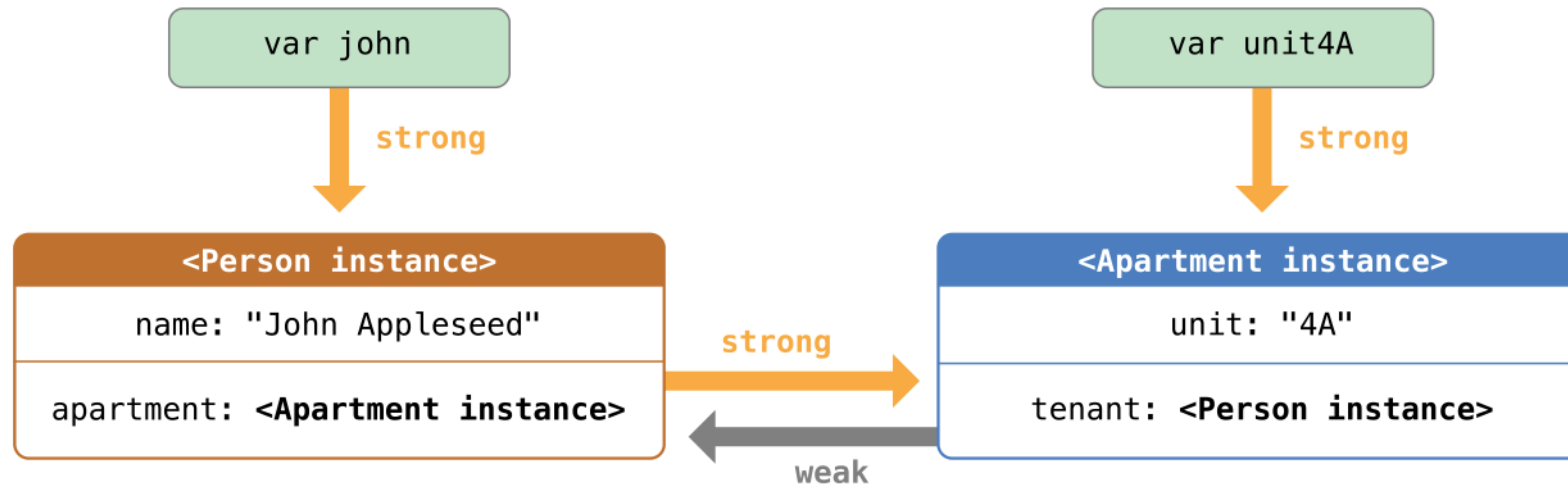
Use case of Weak References



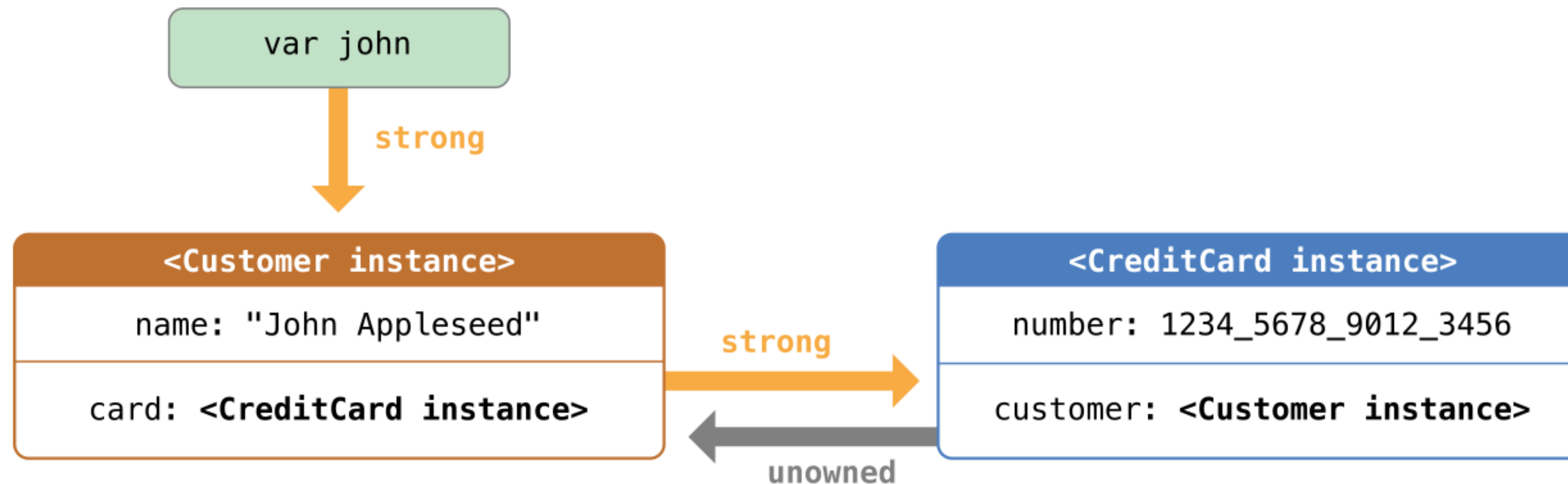
Use case of Weak References



Use case of Weak References



Use case of Unowned References



gitignore

gitignore

- A file specifies intentionally untracked files that Git should ignore.
  https://git-scm.com/docs/gitignore
- github.com/**github**/**gitignore**
- Files of your IDE preferences, temporarily files, built products and secure credentials.
  It may also contains shared content from CocoaPods and Carthage.

OS X development environment

OS X development environment

- In Linux, we have yum and apt-get to install and manage applications, components, and development libraries.
- In OS X, we have homebrew to for development libraries.
  Applications and components are provided from Apple.
- Homebrew installs packages in user domain (*/usr/local*), and hence you don’t need root privilege when using those packages.
  For example, you can install a local copy of Python instead of systems. Also you can install a MySQL db and remove it without leaving trashes on your Mac.

OS X development environment

- Visit http://brew.sh to install and use homebrew.
- We call packages in brew as “formula”.
- `brew install …`
- `brew update && brew upgrade`

- python
- python3
- ruby
- node
- golang
- vim
- bash
- git
- xctool
- swiftlint
- carthage
- youtube-dl

How do Xcode build your apps?

Scheme

Targets

IssueBox - iOS 〉 iPhone 6s Plus

IssueBox | Build IssueBox - iOS: **Succeeded** | 4/5/16 at 11:52 PM

IssueBox M
IssueBox iOS
AppDelegate.swift ?
Main.storyboard ?
Assets.xcassets ?
LaunchScr...toryboard ?
Info.plist ?
IssueBoxCore
Info.plist
module.modulemap
Users.swift A
IssueBoxCoreTests
IssueBoxC...Tests.swift ?
Info.plist ?
Frameworks
Carthage
iOS

IssueBox

General | Capabilities | Resource Tags | Info | Build Settings | Build Phases

PROJECT
IssueBox

TARGETS
IssueBox iOS
IssueBoxCore iOS
IssueBoxCoreTests iOS

Basic | All | Combined | Levels | +

Architectures

Setting	IssueBox iOS
Additional SDKs	
Architectures	Standard architectures (armv7, arm
Base SDK	Latest iOS (iOS 9.3)
Build Active Architecture Only	<Multiple values>
Debug	Yes
Release	No
Supported Platforms	iOS
Valid Architectures	arm64 armv7 armv7s

Assets

Setting	IssueBox iOS
Asset Pack Manifest URL Prefix	

Schemes and Targets

- A target specifies a product to build and contains the rules for building the product from a set of files in a project or workspace. Projects can contain one or more targets, each of which produces one product.
- A scheme defines a collection of targets to build, a configuration to use when building, and a collection of tests to execute. You can many schemes, but only one can be active at a time.

Build Settings (Compile/Linker flags)

General | Capabilities | Resource Tags | Info | Build Settings | Build Phases | Build Rules

Basic | All | Combined | Levels

PROJECT
- IssueBox

TARGETS
- IssueBox iOS
- IssueBoxCore iOS
- IssueBoxCoreTests iOS

Architectures

Setting	IssueBox iOS
Additional SDKs	
Architectures	Standard architectures (armv7, arm64) - $(ARCHS_STANDARD)
Base SDK	Latest iOS (iOS 9.3)
Build Active Architecture Only	<Multiple values>
Debug	Yes
Release	No
Supported Platforms	iOS
Valid Architectures	arm64 armv7 armv7s

Assets

Setting	IssueBox iOS
Asset Pack Manifest URL Prefix	
Embed Asset Packs In Product Bundle	No
Enable On Demand Resources	Yes
On Demand Resources Initial Install Tags	
On Demand Resources Prefetch Order	

Build Locations

Setting	IssueBox iOS

Identity and Type
Name Issue
Location Rela
Full Path /User

Project Document
Project Format Xcod
Organization Issue
Class Prefix

Text Settings
Indent Using Spac
Widths

Source Control
Repository sourc
Type Git
Current Branch maste
Version --
Status Modif

Build Phases (Dependencies, Files, and Libraries)

IssueBox | Build IssueBox - iOS: **Succeeded** | 4/5/16 at 11:52 PM

IssueBox

General | Capabilities | Resource Tags | Info | Build Settings | Build Phases | Build Rules

PROJECT
- IssueBox

TARGETS
- IssueBox iOS
- IssueBoxCore iOS
- IssueBoxCoreTests iOS

Target Dependencies (1 item)
- IssueBoxCore iOS (IssueBox)

Compile Sources (1 item)

Name	Compiler Flags
AppDelegate.swift ...in IssueBoxiOS	

Link Binary With Libraries (2 items)

Name	Status
ObjectMapper.framework	Required

Identity and Type

Project Document

Text Settings

Source Control

Adding 3rd-party libraries

- Some libraries requires special compiler and linker settings.
- Some libraries depend on another 3rd-party libraries.
- Tracking versions of 3rd-party libraries is not easy.
- Use package managers to solve these problems.

Package Managers

Package Managers

- CocoaPods
  Popular and oldest package manager. Easy to learn and use but the whole manager is very heavy.

- Carthage
  Simple and decentralized dependency manager, but requires more sophisticated experience of using Xcode.

- Swift Package Manager
  Official supported by the Swift language, but still in development. (3.0)

Open Source Packages

- Choose packages which is still active, has more “stars”, well tested, and have fully (highly) documented.
- Not all open-source packages are “free” to use.
  You have to follow the policy and guidelines about using it.
- http://www.openfoundry.org/tw/comparison-of-licenses
- Fire an Issue or submit a Pull Request (PR) to contribute to the open source community when you find problems or have a better idea.
  You could also find the solution of your problem by reading old issue posts.

Semantic Versioning (Semver)

- Versioning “x.y.z” as “MAJOR.MINOR.PATCH”
- MAJOR version changes when the package makes incompatible and breaking API changes
- MINOR version is changed when the package adds functionality in a backwards-compatible manner
- PATCH version means the package makes backwards-compatible bug fixes.

Version Operators (CocoaPods)

- > 0.7 - Any version higher than 0.7
- >= 1.0 - Version 1.0 and any higher version
- < 3.0 - Any version lower than 3.0
- <= 2.2 - Version 2.2 and any lower version
- ~> 1.1.2 - Version 1.1.2 and up to 1.2, *not including* 1.2
- ~> 3.2 - Version 3.2 and the up to 4.0, *not including* 4.0

- CocoaControl
- Twitter & RSS feeds
- Developer blogs and communities
  objc.io/issues & nshipster.com & github.com/kilimchoi/engineering-blogs
- Github Trends, Awesome lists, and stars of others
  github.com/trending/swift & github.com/trending/objective-c
  github.com/sindresorhus/awesome & github.com/vsouza/awesome-ios
- StackOverflow

Demo: Use CocoaPods & ObjectMapper

JSON

- JSON (*JavaScript Object Notation*) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate.
- Basic elements are “strings”, “numbers”, “booleans”, and “null”. Like `"string"`, `42`, `true`, `false`, and `null`.
- Collection JSON elements are “dictionaries” and “arrays”. Like `{"key": "value"}` and `["element", 42]`