

курс

development

12 недель

# iOS: разработка приложений с 0

r\_d

iOS

программа курса

20 занятий

1

Swift: начало

2

ООП: основы

3

Создание iOS-приложения в Xcode

4

Создание  
интерфейса iOS-  
приложения

5

Динамические  
интерфейсы, часть 1

6

Динамические  
интерфейсы, часть 2

iOS

программа курса

20 занятий

7

Динамические  
интерфейсы, часть 3

8

Навигация в  
приложении, часть 1

9

Навигация в  
приложении, часть 2

10

Анимации в iOS

11

Работа с памятью  
в iOS

12

Многозадачность  
в iOS, часть 1

iOS

программа курса

20 занятий

13

Многозадачность в  
iOS, часть 2

14

Дебаг  
iOS-приложения

15

Тестирование

16

Хранение данных  
в приложении

17

Работа с сетью  
в приложении

18

Сборка приложения

iOS

программа курса

20 занятий

19

Современные  
архитектуры для  
iOS приложений

20

Защита курсовых  
проектов

# Работа с памятью в iOS

- Общие принципы memory management
- Автоматический подсчёт ссылок (ARC)
- Структуры и классы в Swift с точки зрения хранения в памяти

# Общие принципы memory management

Управление памятью — целый набор механизмов, которые позволяют контролировать доступ программы к оперативной памяти вашего девайса.

# Общие принципы memory management

Для чего используется ОЗУ?

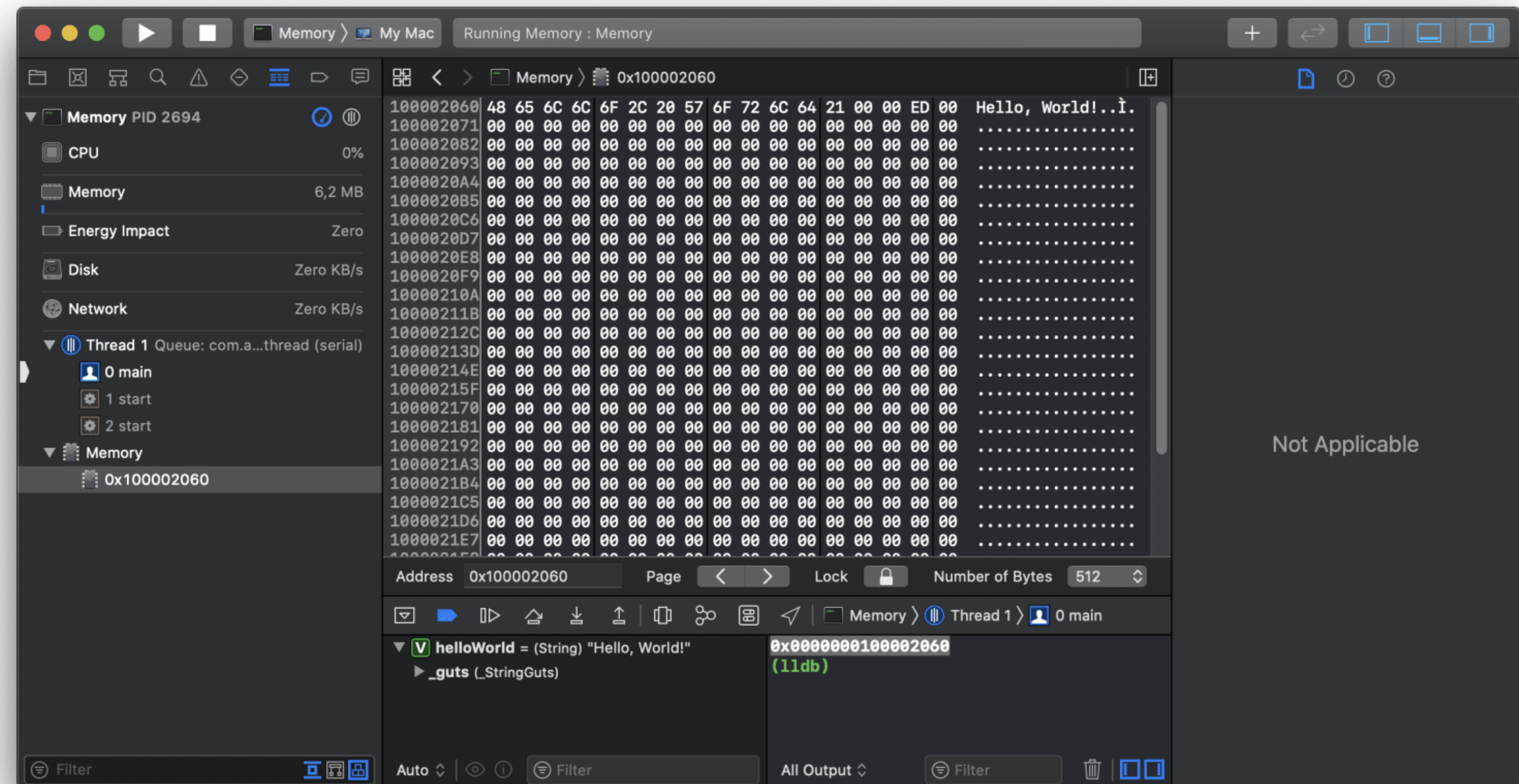
- Программа загружает свой собственный байт-код для выполнения.
- Хранит значения переменных / структур данных, которые используются в процессе работы.
- Загружает внешние модули, необходимые в процессе выполнения.
- Хранит различные ресурсы — изображения, текстовые файлы, аудио/видео контент и многое другое.



# Общие принципы memory management

Кроме загрузки своего собственного байт-кода, программа во время выполнения использует 2 области в ОЗУ:

- стек (stack)
- куча (heap)



# Stack

Стек используется для статического выделения памяти. Он организован по принципу «**последним пришёл — первым вышел**» (LIFO).

Можно представить стек как стопку книг — разрешено взаимодействовать только с самой верхней книгой: прочитать её или положить на неё новую.

## Stack

Static in memory and allocation happens only during compile time.

<Struct>

<Bool>

<Int>

# Stack

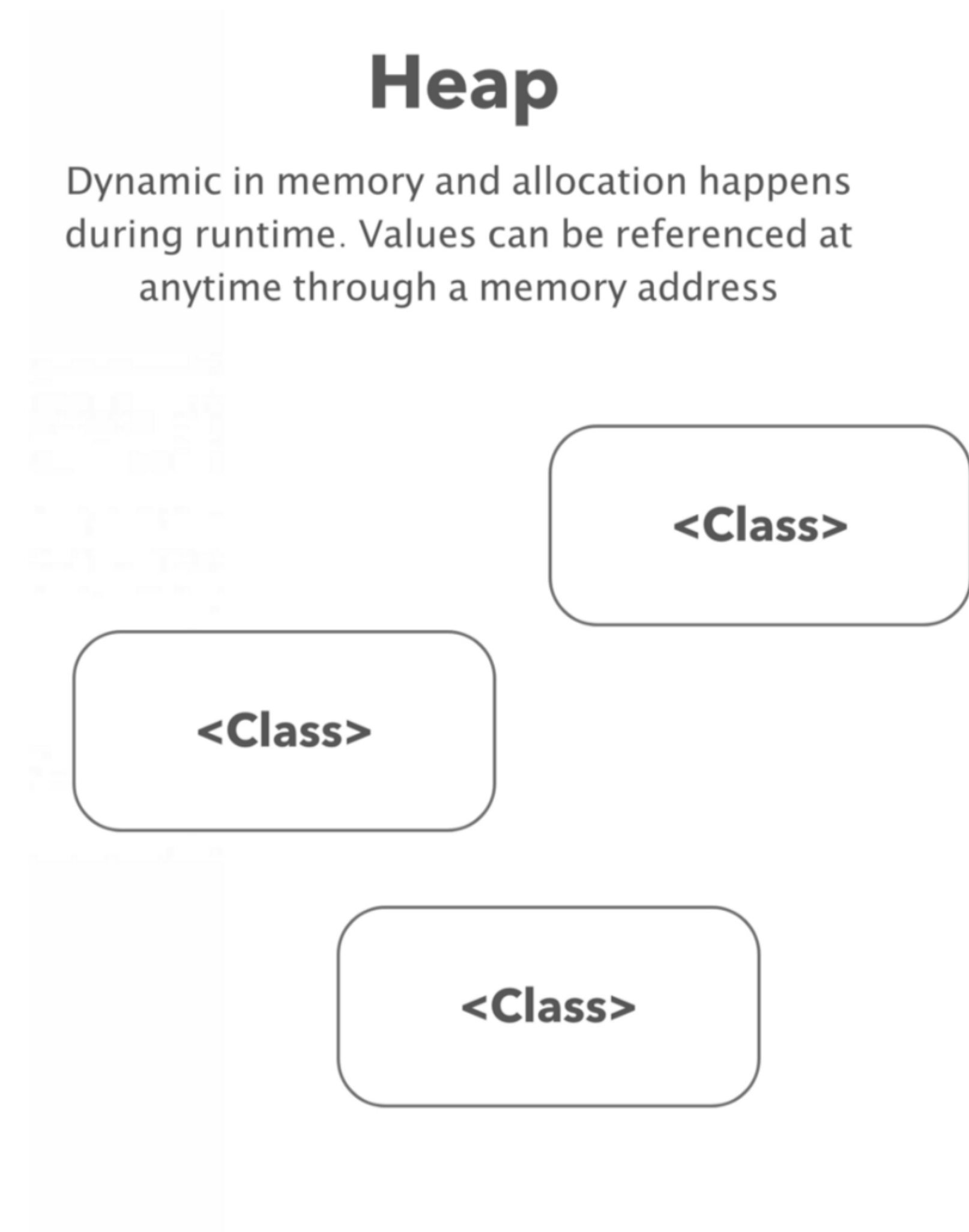
- Управление стековой памятью простое и прямолинейное; оно выполняется операционной системой.
- В стеке обычно хранятся данные вроде локальных переменных и указателей.
- В большинстве языков существует ограничение на размер значений, которые можно сохранить в стек.
- Каждый поток многопоточного приложения имеет доступ к своему собственному стеку.

# Stack

- Существует ограничение: данные, которые предполагается хранить в стеке, обязаны быть конечными и статичными — их размер должен быть известен ещё на этапе компиляции.
- При работе со стеком есть вероятность получать ошибки переполнения стека (`stack overflow`), так как максимальный его размер строго ограничен.
- Например, ошибка при составлении граничного условия в рекурсивной функции совершенно точно приведёт к переполнению стека.

# Heap

- Куча используется для динамического выделения памяти, однако, в отличие от стека, данные в куче первым делом требуется найти с помощью «оглавления».
- Можно представить, что куча это такая большая многоуровневая библиотека, в которой, следуя определённым инструкциям, можно найти необходимую книгу.





# Heap

- В куче хранятся данные динамических размеров, например, список, в который можно добавлять произвольное количество элементов.
- Типичные структуры данных, которые хранятся в куче — это **глобальные переменные** (они должны быть доступны для разных потоков приложения, а куча как раз общая для всех потоков), ссылочные типы, такие как строки или ассоциативные массивы, а так же другие сложные структуры данных.

# Heap

- При работе с кучей можно получить ошибки выхода за пределы памяти (out of memory), если приложение пытается использовать больше памяти, чем ему доступно.
- Размер значений, которые могут храниться в куче, ограничен лишь общим объёмом памяти, который был выделен операционной системой для программы.

# Heap

- Вследствие динамической природы, куча нетривиальна в управлении и с ней возникает большинство всех проблем и ошибок, связанных с памятью.
- Способы решения этих проблем предоставляются языками программирования.



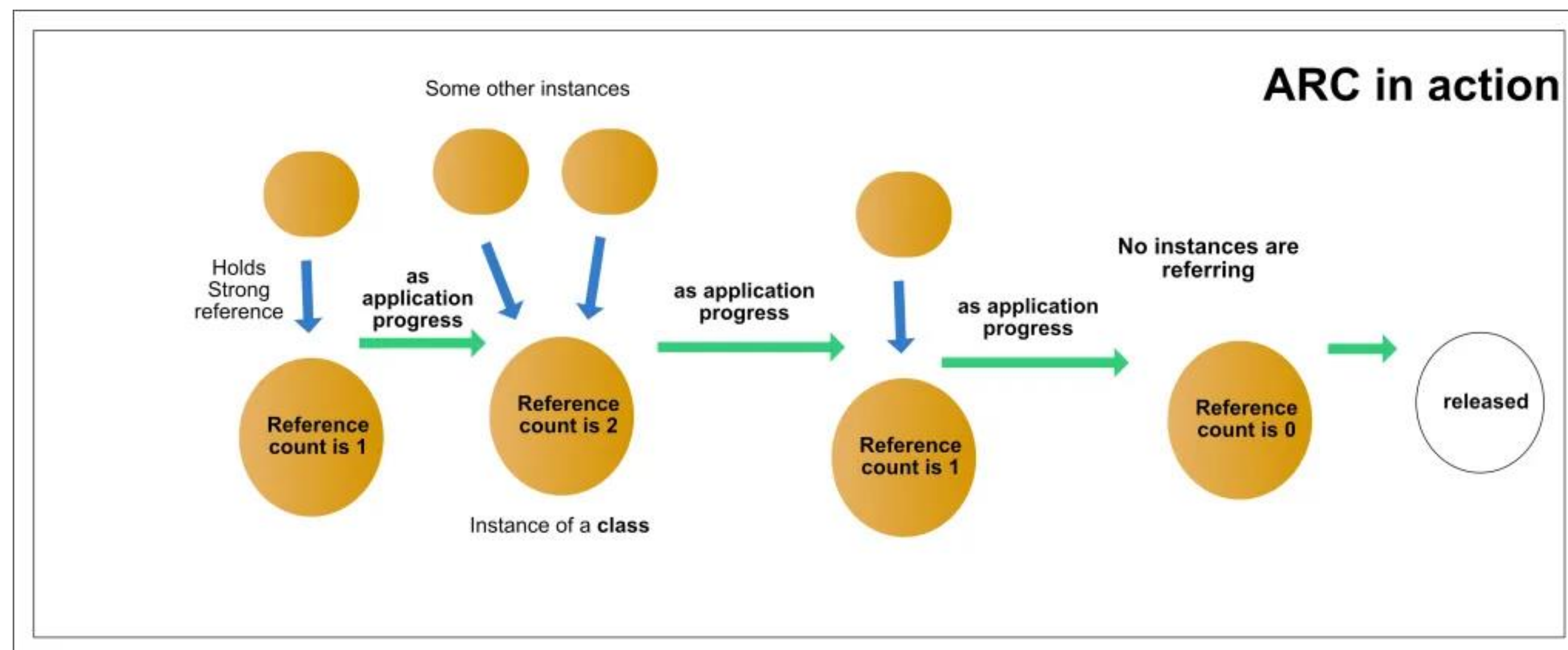
# Общие принципы memory management

Наиболее распространенные методы работы с памятью в ЯП:

- ручное управление памятью (C, C++)
- сборщик мусора (Java, C#, Go, Python)
- подсчёт ссылок (Objective-C, Swift)

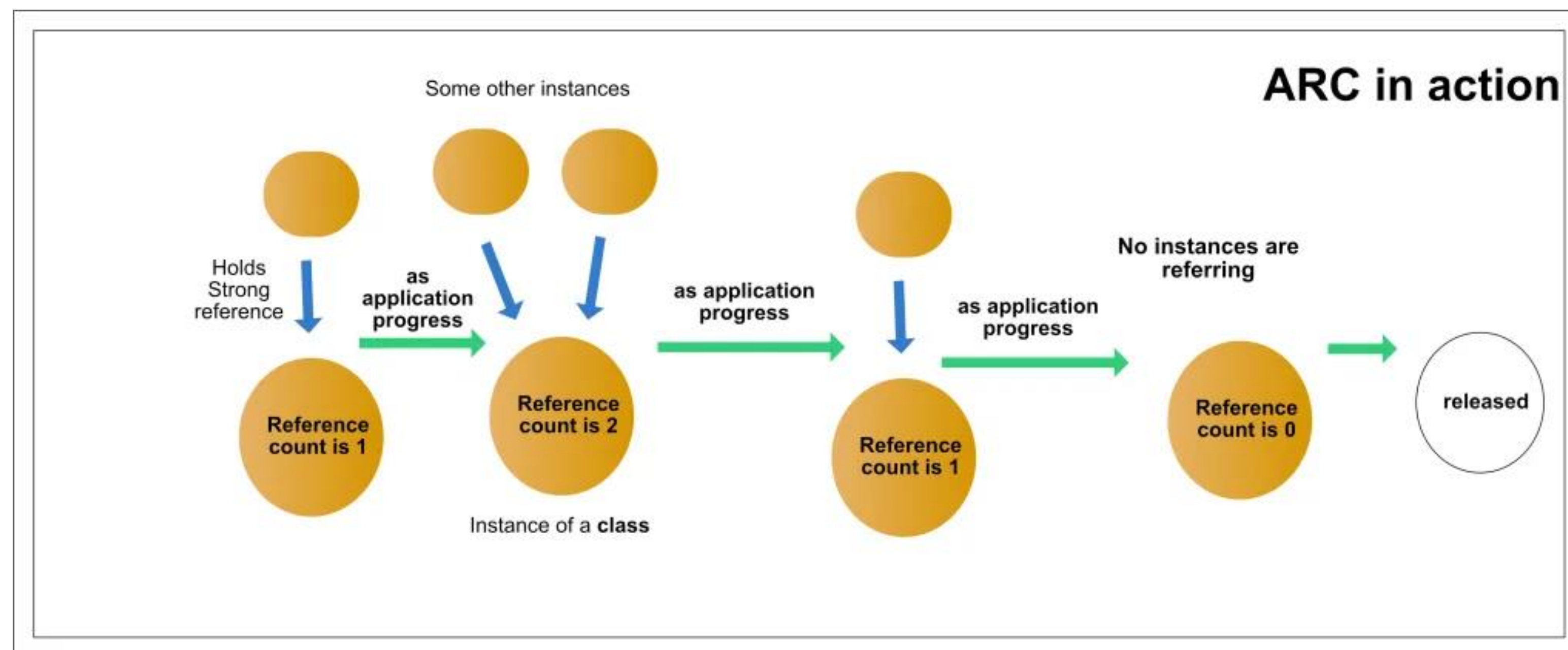
# Автоматический подсчёт ссылок (ARC)

Каждый раз, когда вы создаете экземпляр класса, ARC выделяет фрагмент памяти для хранения информации этого экземпляра. Этот фрагмент памяти содержит информацию о типе экземпляра, о его значении и любых свойствах хранения, связанных с ним.



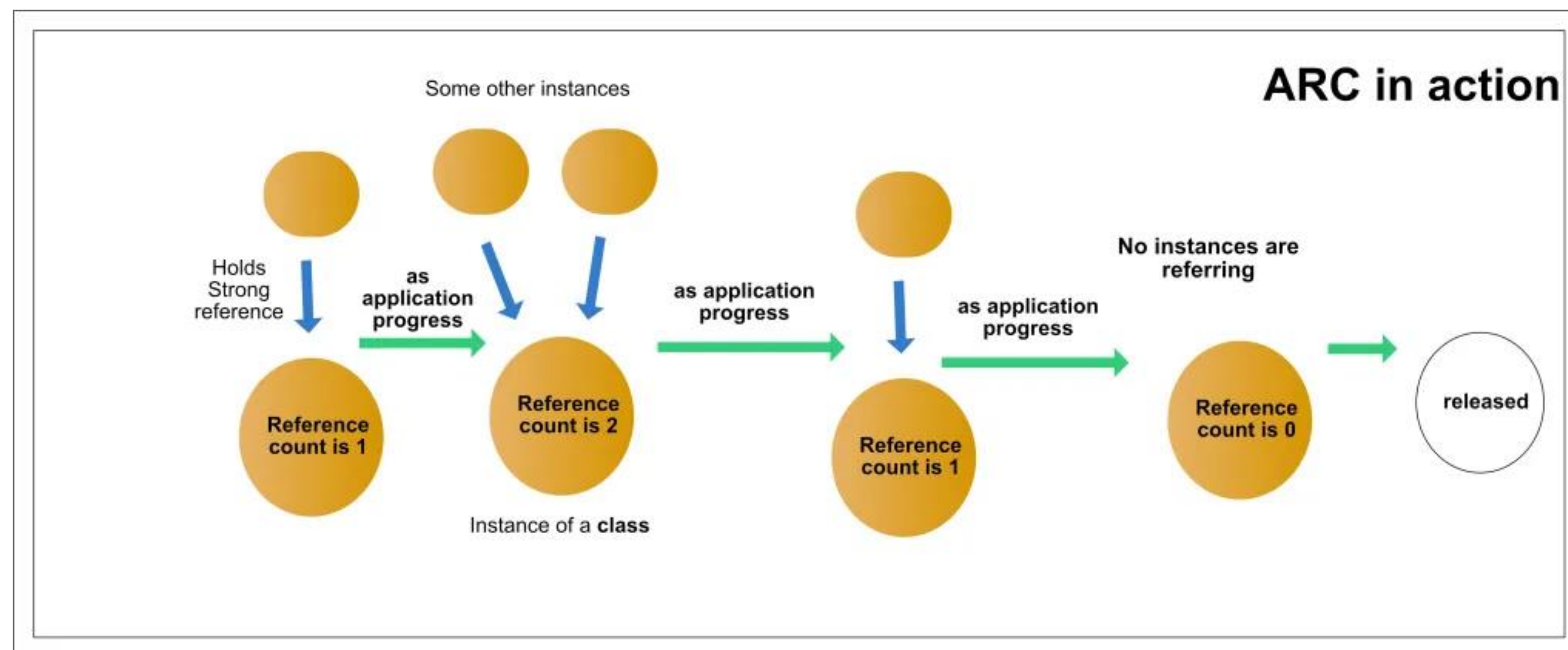
# Автоматический подсчёт ссылок (ARC)

Дополнительно, когда экземпляр больше не нужен, ARC освобождает память, использованную под этот экземпляр, и направляет эту память туда, где она нужна. Это своего рода гарантия того, что ненужные экземпляры не будут занимать память.



# Автоматический подсчёт ссылок (ARC)

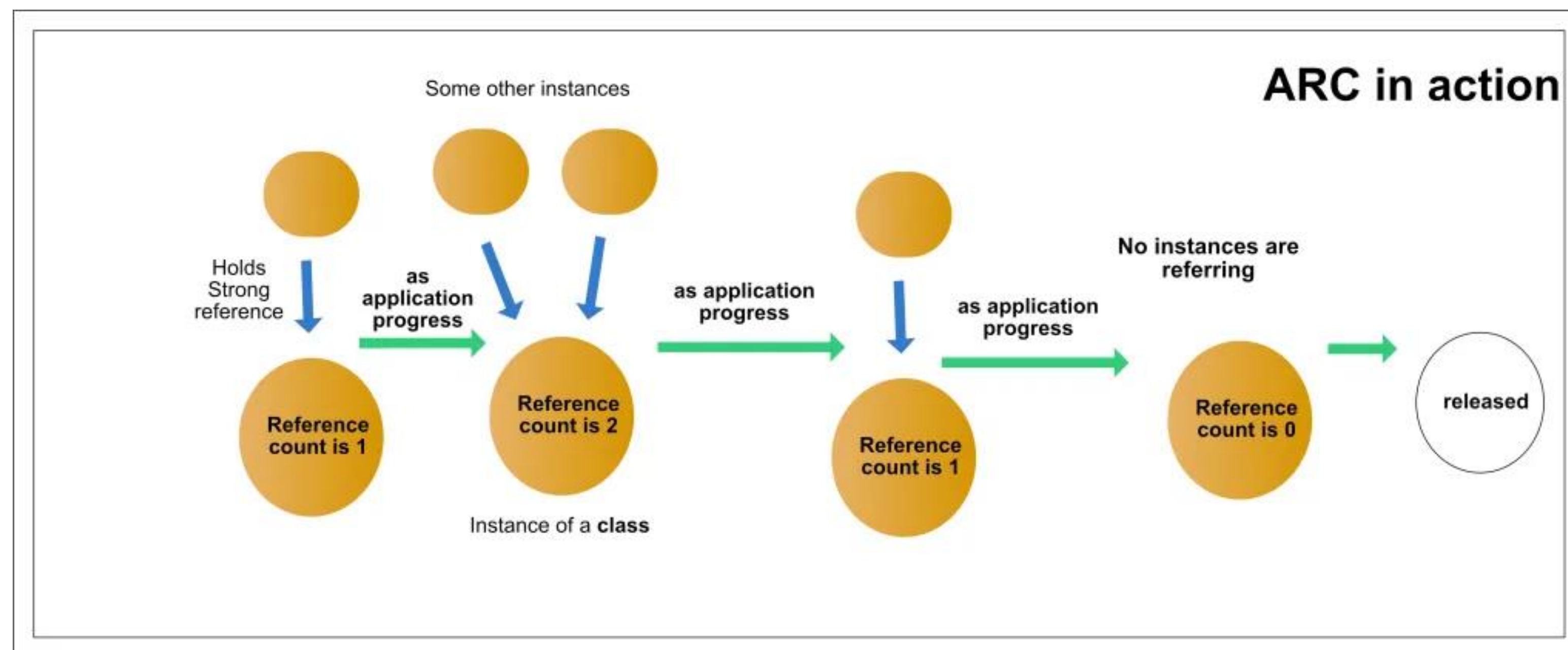
Однако, если ARC освободит память используемого экземпляра, то доступ к свойствам или методам этого экземпляра будет невозможен. Если вы попытаете получить доступ к этому экземпляру, то ваше приложение скорее всего выдаст ошибку и будет остановлено.





# Автоматический подсчёт ссылок (ARC)

Для того, чтобы нужный экземпляр не пропал, ARC ведет учет количества свойств, констант, переменных, которые ссылаются на каждый экземпляр класса. ARC не освободит экземпляр, если есть хотя бы одна активная ссылка.



# Автоматический подсчёт ссылок (ARC)

- Для того чтобы учёт ссылок работал, каждый раз, как вы присваиваете экземпляр свойству, константе или переменной создается strong reference (сильная ссылка) с этим экземпляром.
- Такая связь называется “сильной”, так как она крепко держится за этот экземпляр и не позволяет ему освободиться до тех пор, пока остаются сильные связи.

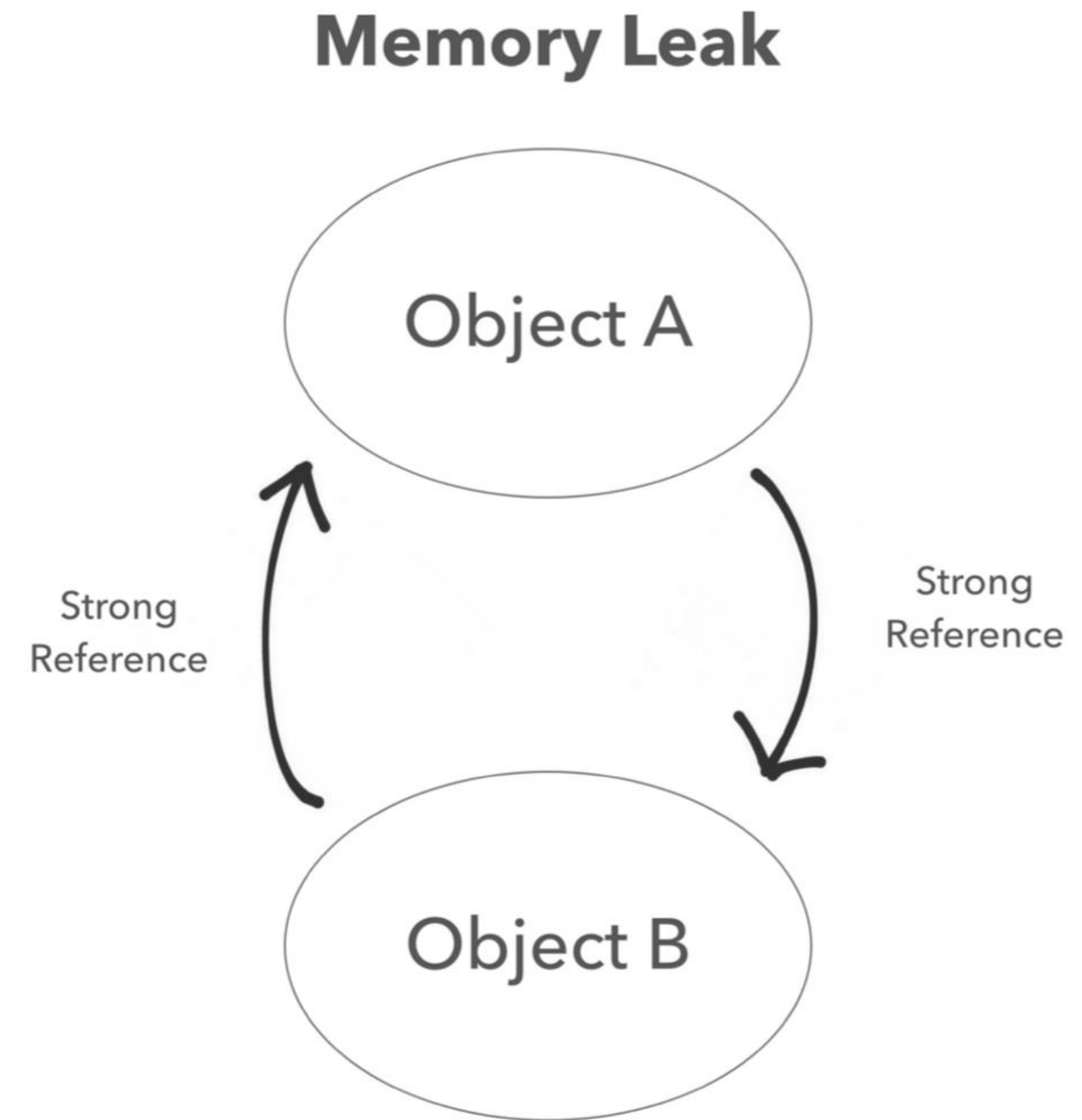
# Автоматический подсчёт ссылок (ARC)

Важно!

- Механизм ARC применим только для экземпляров класса.
- Структуры и перечисления являются типами значений (value types), а не ссылочными типами (reference types), и они не хранятся и не передают свои значения по ссылке.

# Автоматический подсчёт ссылок (ARC)

- Однако, может случиться такая ситуация, когда объект A содержит сильную ссылку на объект B, а объект B - сильную ссылку на объект A.
- В таком случае экземпляр класса A никогда не будет иметь нулевое число сильных ссылок. Объект не освободиться из памяти, что в свою очередь создаст цикл сильных ссылок (reference cycle).





# Автоматический подсчёт ссылок (ARC)

- Чтобы избежать такой проблемы и разорвать цикл сильных ссылок, нужно один из объектов, который хранится в реализации класса, объявить как слабую ссылку (weak reference).

```
class A {  
    let name: String  
    init(name: String) { self.name = name }  
    var objectB: B?  
    deinit { print("\(name) деинициализируется") }  
}
```

```
class B {  
    let unit: String  
    init(unit: String) { self.unit = unit }  
    weak var objectA: A?  
    deinit { print("Apartment \(unit) деинициализируется") }  
}
```

# Структуры, классы и memory management

- Структуры и перечисления — типы значения (value types).
- Классы — ссылочные типы (reference types)

# Структуры, классы и memory management

- Тип значения - это тип, значение которого копируется, когда оно присваивается константе или переменной, или когда передается функции.
- Вы уже достаточно активно использовали типы на протяжении предыдущих глав. Но факт в том, что все базовые типы Swift - типы значений и реализованы они как структуры.
- Все структуры и перечисления - типы значений в Swift. Это значит, что любой экземпляр структуры и перечисления, который вы создаете, и любые типы значений, которые они имеют в качестве свойств, всегда копируются, когда он передается по вашему коду.

# Структуры, классы и memory management

- В отличие от типа значений, ссылочный тип не копируется, когда его присваивают переменной или константе, или когда его передают функции. Вместо копирования используется ссылка на существующий экземпляр.
- Так как классы являются ссылочными типами, то есть возможность сделать так, чтобы несколько констант и переменных ссылались на один единственный экземпляр класса. (Такое поведение не применимо к структурам и перечислениям, так как они копируют значение, когда присваиваются константам или переменным или передаются функциям.)


# Структуры, классы и memory management

Иногда бывает полезно выяснить ссылаются ли две константы или переменные на один и тот же экземпляр класса. Для проверки этого в Swift есть два оператора тождественности:

- Идентичен (===)
- Не идентичен (!==)

# Структуры, классы и memory management

Можно использовать эти операторы для проверки того, ссылаются ли две константы или переменные на один и тот же экземпляр:



```
if objectA === alsoObjectA {  
    print("objectA and alsoObjectA refer to the same ClassA instance.")  
}  
// Выведет "objectA and alsoObjectA refer to the same ClassA instance."
```

# Структуры, классы и memory management

- Обратите внимание, что «идентичность» (в виде трех знаков равенства, или ===) не имеет в виду «равенство» (в виде двух знаков равенства, или ==).
- Идентичность или тождественность значит, что две константы или переменные ссылаются на один и тот же экземпляр класса.
- Равенство значит, что экземпляры равны или эквивалентны в значении в самом обычном понимании «равны».

спасибо

задавайте вопросы