Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(государственный университет)
ФАКУЛЬТЕТ РАДИОТЕХНИКИ И КИБЕРНЕТИКИ
КАФЕДРА ТЕОРЕТИЧЕСКОЙ И ПРИКЛАДНОЙ ИНФОРМАТИКИ

# ИССЛЕДОВАНИЕ ПРОИЗВОДИТЕЛЬНОСТИ ЯЗЫКА SWIFT

Выпускная квалификационная работа

Направление подготовки: 03.03.01 Прикладные математика и физика

Выполнил:
студент 311 группы: _____ Матвеев Иван Сергеевич

Научный руководитель:_____ Соболев Артемий Анатольевич

Москва
2017

# Оглавление

Оглавление....2
1. Введение....3
2. Актуальность....5
3. Постановка задачи....6
4. Внутреннее устройство основных примитивов Swift....7
5. Инструменты....13
6. Исследование....14
7. Результаты....27
8. Дальнейшее развитие....29
9. Приложение....31
10. Список литературы....35

# 1. Введение

При появлении новых языков программирования возникает задача сравнить их с существующими. Так Objective-C многие годы использовался Apple как основной язык для своих разработок, но в 2014 году компания представила новый язык - Swift. Возникает вопрос - действительно ли Swift так хорош, что компания решила переписать на нем большую часть кода своих продуктов?

Objective-C является расширением языка Си, в который были добавлены новые возможности для объектно-ориентированного подхода программирования. Создателями Objective-C являются Брэд Кокс и Том Лав. Они начали работать над ним в начале 1980-х годов.

Язык использует объектную модель Smalltalk. В этой модели классы являются полноценными объектами, поддерживается интроспекция и динамическая обработка сообщений. Любому объекту можно послать любое сообщение. Объект может вместо обработки сообщения переслать его другому объекту для обработки (делегирование), в частности, так можно реализовать распределённые (то есть находящиеся в различных адресных пространствах и даже на разных компьютерах) объекты. В Objective-C во время исполнения можно узнать тип объекта, список методов, проверить является ли он потомком заданного объекта, поддерживает ли протокол и так далее. Objective-C полностью совместим с языком программирования Си. Компания Apple долгое время использовала его как основной язык программирования для разработки своих продуктов.

Swift стал самым быстрорастущим языком программирования в истории. Согласно TIOBE сейчас он занимает 14 позицию в мировом рейтинге языков программирования. Он заимствовал идеи из Objective-C, Rust, Haskell, Ruby, Python, C#, CLU и других языков. Его создатель - Крис Латтнер. Над Swift он начал работать летом 2010 года, к нему присоединилось множество разработчиков. Apple держала это в секрете полтора года. Компания позиционирует этот язык как более быстрый и

эффективный способ создания программ для iPhone, iPad, Mac, Apple Watch и Apple TV. Google объявила о намерениях сделать Swift так называемым «первым языком» для Android. Крис Латтнер надеется что наступит день, когда Swift станет самым популярным языком в мире, будет использоваться не только для создания приложений, но и в системном программировании.

## 2. Актуальность

Apple одна из крупнейших компаний по производству смартфонов, ноутбуков и другой электроники, но в отличие от большинства других компаний Apple сама пишет программное обеспечения для своих устройств.

Apple предоставляет инфраструктуру, для простого и удобного написания быстрых и эффективных по потребляемой энергии приложений под iOS или macOS. В эту инфраструктуру входит среда разработки Xcode, языки Swift и Objective-C, а также различные библиотеки и фреймворки. Также ежегодно в Калифорнии проводится Worldwide Developers Conference (WWDC) - всемирная конференция для разработчиков на платформах Apple. На этих конференциях анонсируют новые продукты, разработки и обновления платформ Apple.

Swift пришел на смену старому и громоздкому Objective-C, который был разработан еще в 1980-х годах. Apple проводит обучающие курсы по Swift, на iPad сделано приложение чтобы дети могли изучать его. На WWDC обновления и улучшения библиотек и фреймворков рассказывают на Swift, а не на Objective-C, поэтому чтобы узнать все новинки нужно хоть немного знать Swift. Он требует меньше кода, более читаемый, безопасный и интерактивный. Но готов ли он стать инструментом разработки в индустрии? Для этого необходимо чтобы Swift был не менее производительным чем Objective-C. Исследованием этого вопроса и занимается эта работа.

Также Swift обладает своими особенностями, которые влияют на производительность. Поэтому чтобу написать быструю и эффективную программу на Swift нужно разобраться как он устроен внутри - как управляет памятью, какими сущностями пользуется. Также немаловажным свойством языка является время компиляции - если оно большое, то разработка приложения, или иного продукта сильно замедляется. Поэтому нужно знать что может замедлить компиляцию.

# 3. Постановка задачи

В этой работе сравниваются языки Swift и Objective-C, а также исследуются способы написания действительно быстрых программ на Swift.

- Сравнение времени работы стандартных алгоритмов сортировки на Swift, Objective-C и C
- Сравнение времени создания и прохода по массиву при полиморфизме основанном на наследовании классов, на протоколах и при использовании шаблонов.
- Тестирование стандартных коллекций элементов Array, Dictionary и Set на Swift и Objective-C

# 4. Внутреннее устройство основных примитивов Swift

Согласно лекции проведенной на WWDC в 2016 году существуют 3 показателя, которые влияют на производительность языка Swift

1) Allocation (выделение памяти) - Heap или Stack

2) Automatic Reference Counting (ARC) (автоматический подсчет сильных ссылок на объект)

3) Method Dispatch (вычисление адреса вызываемой функции во время исполнения - dynamic, во время компиляции - static)

**Allocation**

Чтобы выделить или освободить память на стеке достаточно соответственно уменьшить или увеличить stack pointer. В то время как при выделении памяти в куче происходит поиск свободного блока памяти. Также появляются накладные расходы из-за много-поточности.

**Automatic Reference Counting**

Согласно лекции WWDC 2016 классы в Swift являются ссылочными объектами - на стеке выделяется указатель, а сам объект создается в куче. Но возникает вопрос когда освобождать память выделенную под объект, так как указатель на него может быть передан в другую функцию или поток. Для решения этой проблемы используют ARC - в каждом объекте класса есть дополнительная переменная которая содержит количество сильных ссылок на объект, и когда она становится равной нулю вызывается метод destroy и память освобождается.

Таким образом когда создается много ссылочных объектов, появляются накладные расходы связанные с увеличением и уменьшением счетчиков ссылок.

Самый плохой случай когда структура (объект который передается по значению) содержит много ссылочных объектов, так как тогда при копировании структуры будет увеличиваться счетчик ссылок каждого ссылочного объекта. Для классов такой проблемы нет, так как они

передаются по указателю, что увеличивает счетчик ссылок только самого объекта, а все счетчики ссылок объектов, которые содержит класс, останутся без изменений.

Automatic reference counting также используется в Objective-C, избавляя разработчика от необходимости написания вызовов retain/release/autorelease.

В отличие от сборщика мусора ARC не реагирует на циклы retain, то есть пара объектов, имеющие сильные ссылки друг на друга не будут удалены из памяти, даже если никто другой на них не ссылается. Разработчику нужно самому разрешать циклы retain, используя слабые ссылки (они не увеличивают счетчик ссылок). Это касается как Objective-C, так и Swift.

**Method Dispatch**

Методы - это функции, ассоциированные с определенным типом, Классы, структуры и перечисления могут определять методы «экземпляра», которые инкапсулируют специфическую функциональность для работы с «экземпляром» заданного «типа». Также «тип» может определить методы, которые ассоциируются с «типом» как таковым.

Случай статического определения адреса вызываемого метода в коде выглядит так:

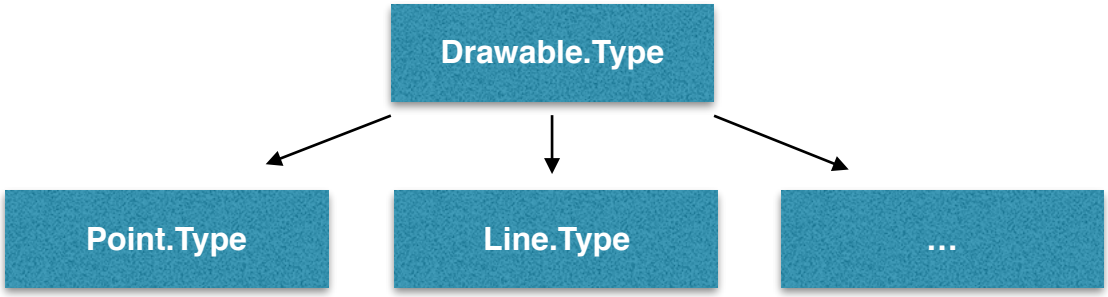
```
struct A {
    static func doSomething() {
        // do something
    }
}

A.doSomething()
```

То есть во время компиляции известно какой метод будет вызван - нет никаких накладных расходов, а также компилятор может применить оптимизации. Например подставить имплементацию метода в код, что убирает накладные расходы на передачу аргументов и на прыжок на реализацию метода (прыжки jmp мешают работе конвейера).

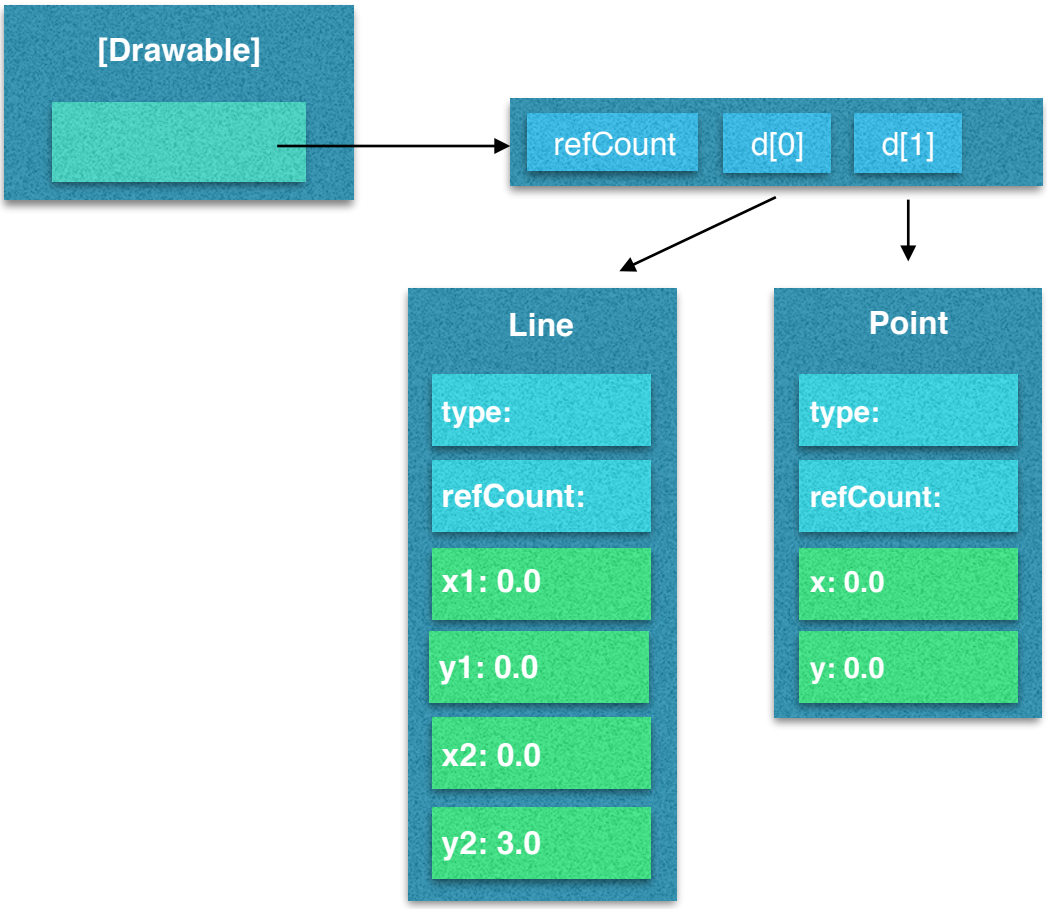
В случае динамического определения адреса метода - то есть выбор метода, который будет вызван, выполняется во время исполнения - невозможны оптимизации компилятора.

Рассмотрим пример с ***полиморфизмом, основанном на наследовании.***

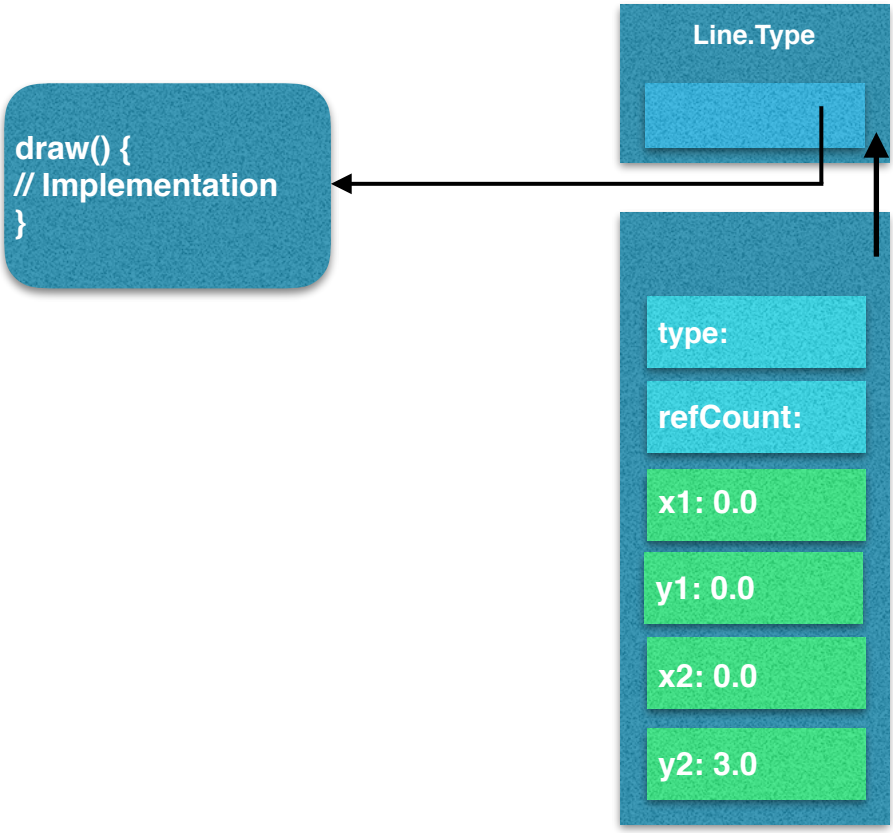
Пусть есть классы Point и Line, которые наследуют класс Drawable, в котором определен метод draw().




Тогда при определении массива типа Drawable, в памяти он будет выглядеть так:



Если у элемента массива вызвать метод draw(), то произойдет еще 2 перехода

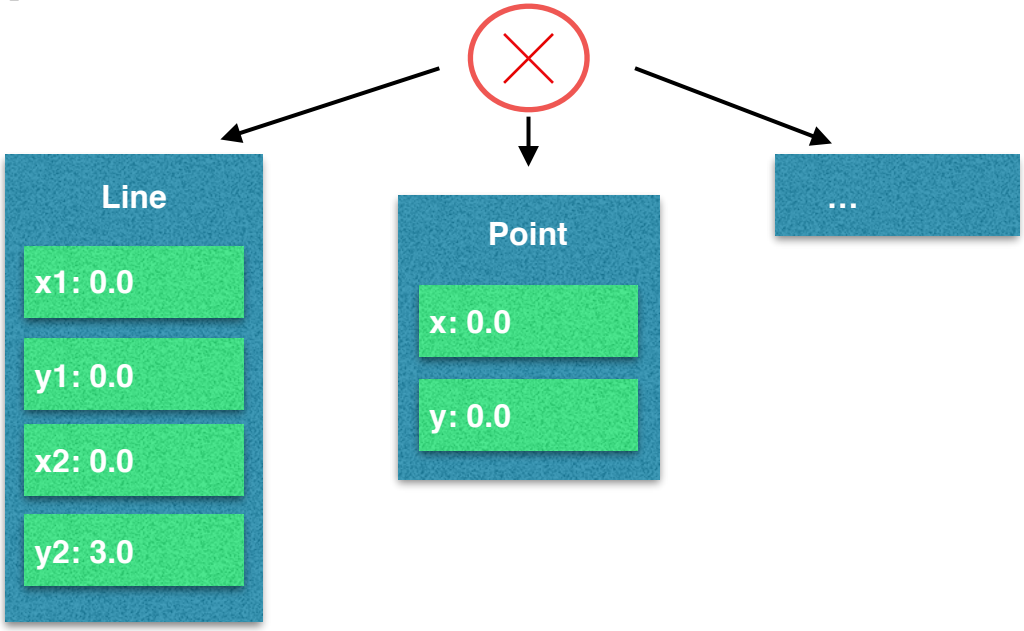



Таким образом сначала нужно прочитать тип объекта, затем перейти на структуру .Type, которая содержит указатели на методы класса, и только после этого вызывается функция.

В случае если класс не предполагает что наследники будут переопределять этот метод, нужно писать final перед определением метода. Также можно отметить весь класс как конечный, написав final class. Тогда компилятор в некоторых случаях может убрать работу dynamic method dispatch, и применить оптимизации доступные при static method dispatch.

Немного по-другому обстоит дело с ***полиморфизмом, основанном на протоколах***. Протокол может поддержать как структура, так и класс. Нужно универсальное хранилище для объектов протокольного типа, для передачи как аргумент в функцию или при создании массива. Для этого используется «контейнер», который будет описан ниже.

Пусть теперь будет протокол Drawable, содержащий метод draw(), и структуры Point и Line.




Теперь при определении массива Drawable, для хранения объектов используется «контейнер» - структура содержащая буфер, размером в 3 слова, а также указатели на Value Witness Table и на Protocol Witness Table. Value Witness Table содержит методы allocate, copy, destruct, deallocate, а Protocol Witness Table содержит методы объекта, требуемые протоколом. Если структура вмещается в буфер, то она там и хранится, в противном случае - в буфере лежит указатель на структуру, которая лежит в куче.

Таким образом при вызове функции draw(), сначала нужно перейти на таблицу Protocol Witness Table и затем там найти нужную функцию.

С этим связана еще одна особенность, так как структура в функцию передается по значению, то сначала создается «контейнер», в нем проставляются указатели на таблицы, и если структура не помещается в 3 слова, то происходит выделение памяти в куче, и копирование туда структуры. При использовании классов дополнительного выделения и освобождения памяти в куче не происходит.

***Шаблоны*** позволяют использовать статический полиморфизм. Также как и в протоколах есть одна реализация, значение передается используя «контейнер», и если структура не помещается в 3 слова, снова происходит выделение памяти в куче, а также невозможны оптимизации как в

динамическом полиморфизме. Но если определение структуры (объекта) и использование шаблонов происходит в одном модуле, то при включении опции компилятора Whole Module Optimization, происходит специализация шаблонов. То есть для каждого типа создается своя реализация функции.

Итого, для специализированных шаблонов при использовании структур:

1) нет выделения памяти в куче.
2) нет счетчика ссылок.
3) статический вызов методов.

Для классов:

1) выделение памяти в куче при создании нового объекта.
2) есть счетчик ссылок.
3) динамический вызов методов (если класс не final).

Если структура большая - передача по значению обходится дорого, то можно в ней создать класс - хранилище, который будет содержать все остальные поля, и в структуре определить метод копирования, что хранилище нужно копировать только если в него идет запись - этот метод называется copy on write.

Вывод: при использовании структур малого размера - помещающихся в 3 64 битных слова - полиморфизм основанный на протоколах дает выигрыш, по сравнению с наследованием, так как нет automatic reference counting и выделения памяти в куче. Если структура больше 3 слов, то нужно использовать наследование классов, так как выделение памяти в куче на каждую передачу аргументов сильно замедляет программу.

Неспециализированные шаблоны по производительности не отличаются от протокольных типов, поэтому шаблоны стоит использовать если есть возможность их специализировать.

# 5. Инструменты

Измерение времени - CACurrentMediaTime()

Для измерения времени операций в структурах данных был использован XCTest - стандартная библиотека предоставляемая Xcode.

Измерения проводились на ноутбуке:

MacBook Air (13-inch, Early 2014)

Процессор 1,4 GHz Intel Core i5

Память 4 Гб, 1600MHz DDR3

# 6. Исследование

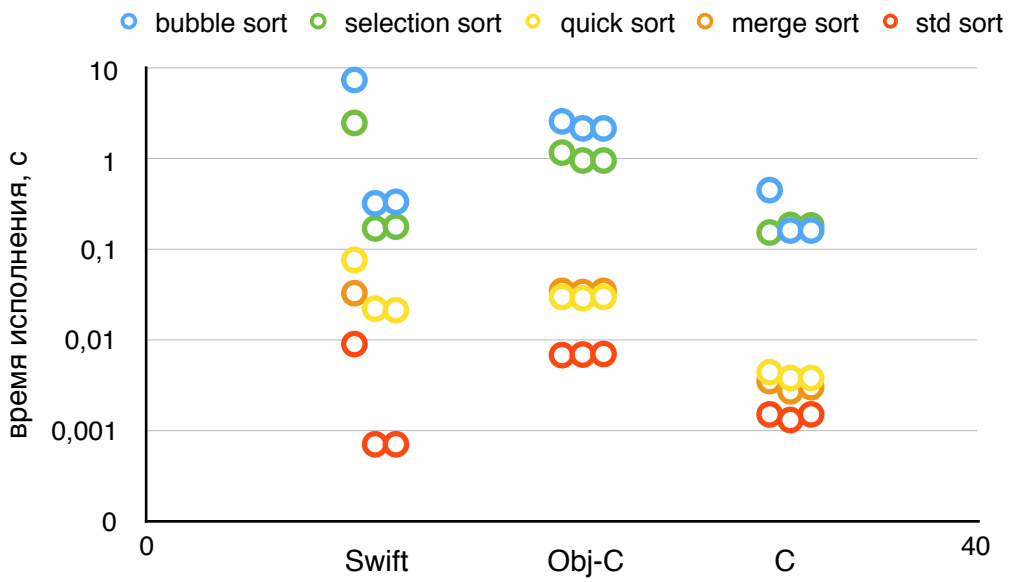
## 6.1 Сортировка

Было написано несколько стандартных алгоритмов сортировки на C, Swift и Objective-C, и измерены времена работы при разных уровнях оптимизации, на массиве с элементами Int размером в 10000, значение времени было усреднено по 100 измерениям.

1) *Сортировка пузырьком* - сравниваются соседние значения и если нужное неравенство не выполняется, то элементы меняются. $O(n^2)$
2) *Сортировка выборкой* - выполняем поиск наименьшего элемента массива, вставляя его в начало, повторяем эти действия для оставшегося массива. $O(n^2)$
3) *Быстрая сортировка* - берется случайный элемент массива, удаляется из массива, и массив разбивается на 2 - в первом те элементы что меньше выбранного, во втором остальные. И к каждому из полученных массивов применяются те же действия, пока размер массива не станет нулевым. $O(n*\log_2 n)$
4) *Сортировка слиянием* - массив делится пополам, и к каждой половине применяется сортировка слиянием, и после происходит слияние отсортированных половинок за линейное время. $O(n*\log_2 n)$
5) Стандартная сортировка, которая предполагается стандартной библиотекой.

| | | bubble sort | selection sort | quick sort | merge sort | std sort |
|---|---|---|---|---|---|---|
| **Swift** | None [-O0] | 7,572 | 2,531 | 0,077 | 0,033 | 0,009 |
| | Fast [-O] | 0,326 | 0,172 | 0,0222 | - | 0,0007 |
| | [-O-wmo] | 0,339 | 0,180 | 0,0214 | - | 0,0007 |
| **Obj-C** | None [-O0] | 2,638 | 1,189 | 0,030 | 0,035 | 0,0068 |
| | Fast [-O3] | 2,205 | 0,972 | 0,029 | 0,034 | 0,0069 |
| | [-Ofast] | 2,203 | 0,971 | 0,030 | 0,035 | 0,007 |
| **C** | None [-O0] | 0,457 | 0,155 | 0,0044 | 0,0035 | 0,0015 |
| | Fastest [-O3] | 0,163 | 0,188 | 0,0038 | 0,0027 | 0,0013 |
| | [-Ofast] | 0,163 | 0,186 | 0,0038 | 0,0030 | 0,0013 |
| **Swift ClassInt** | [-O-wmo] | 5,747 | 5,730 | 1,685 | - | 0,0067 |

Значения в таблице представлены в секундах.




На графике для каждого языка, для каждого алгоритма сортировки есть 3 точки - сначала при отсутствии оптимизации, потом при средней и при максимальной оптимизациях - соответственно слева на право.

На Swift при переходе с нулевого уровня оптимизации на оптимизацию внутри каждого файла (средний уровень оптимизации) время работы уменьшается в десятки раз, и в сортировке выборкой он близок к C, в то время как на Objective-C время практически не меняется. Здесь явно видно

что Swift быстрее чем Objective-C, а C быстрее чем Swift, разве что при использовании стандартной сортировки Swift в 2 раза быстрее. Но если сделать класс - обертку над Int - ClassInt, то время работы самоличных алгоритмов сильно замедляется, а стандартная сортировка в Swift работает с той же скоростью, как и в Objective-C.

### 6.2 Полиморфизм

1) *Полиморфизм основанный на наследовании* - пусть есть класс Weighty, который содержит метод centerMass() - возвращает координаты центра масс и суммарную массу объекта. От него наследуется 2 класса - Point и Line.

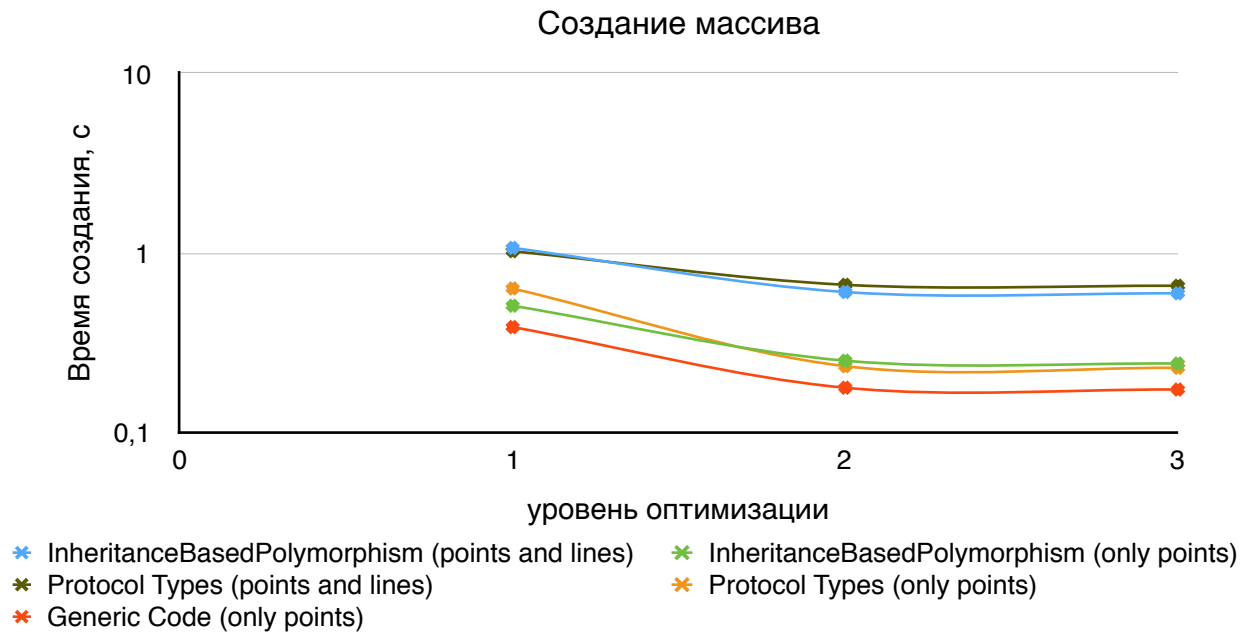
Далее создаю массив типа Weighty, чередуя Point и Line (или только Point). И делая линейный проход по массиву считаю суммарный центр масс.

2) *Полиморфизм основанный на протоколах* - все тоже самое, только Weighty теперь протокол, а Point и Line - структуры.

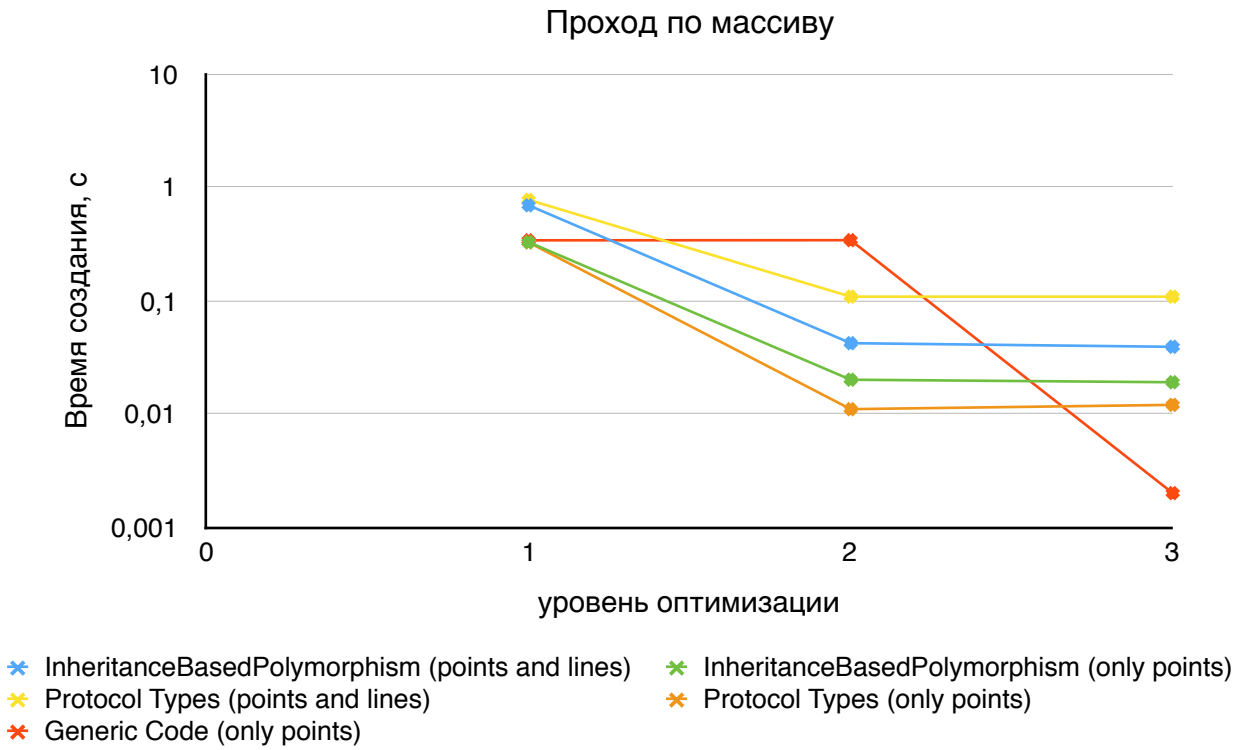
3) *В случае шаблонов* требуется чтобы на вход функции подавался определенный тип, соответственно был использован массив Point-ов.

Значения в таблице представлены в секундах.

| | | | None [-Onone] | Fast [-O] | [-O wmo] |
|---|---|---|---|---|---|
| **Inheritance Based Polymorphism** | points and lines | create array | 1,071 | 0,609 | 0,599 |
| | | go through | 0,690 | 0,042 | 0,039 |
| | only points | create array | 0,511 | 0,253 | 0,244 |
| | | go through | 0,327 | 0,020 | 0,019 |
| **Protocol Types** | points and lines | create array | 1,027 | 0,668 | 0,660 |
| | | go through | 0,771 | 0,108 | 0,108 |
| | only points | create array | 0,637 | 0,236 | 0,231 |
| | | go through | 0,325 | 0,011 | 0,012 |
| **Generic Code** | only points | create array | 0,389 | 0,179 | 0,175 |
| | | go through | 0,339 | 0,340 | 0,002 |




По графику видно, что времена создания массива при наследовании и при использовании протокола практически не отличаются, а шаблоны немного, но выигрывают.

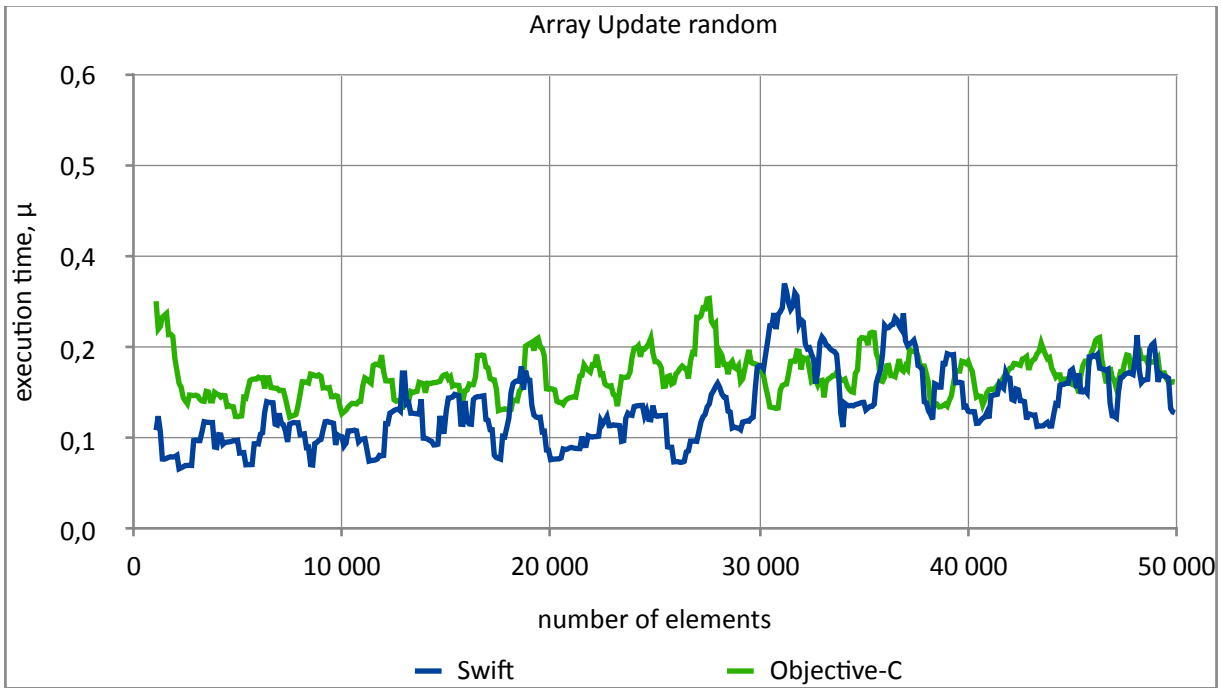



Но по времени прохода по массиву при наличии разных типов элементов в массиве, выигрывает наследование, в то время как для одного типа лучше использовать протоколы. И явно видно что при включенной опции Whole Module Optimization шаблоны специализируются и работают в 6-9 раз быстрее.

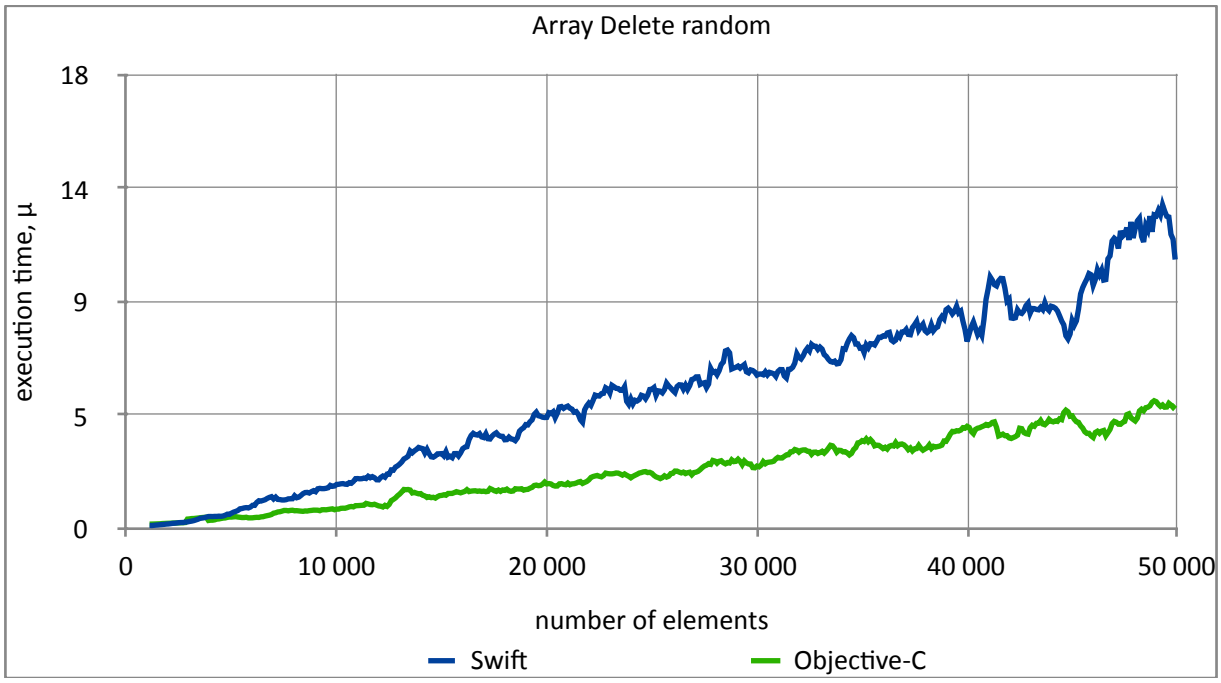
### 6.3 Измерение времени операций структур данных в Swift и в Objective-C

Измеряется время исполнения операций стандартных структур данных: Array, Dictionary и Set, а именно добавление, удаление, чтение, обновление случайного элемента и проверка что элемент присутствует в структуре данных. Измерение было усреднено по 10 запускам для каждого количества элементов в структуре. Количество элементов изменяется с шагом 100, до 50 000. Тип элемента - Int.

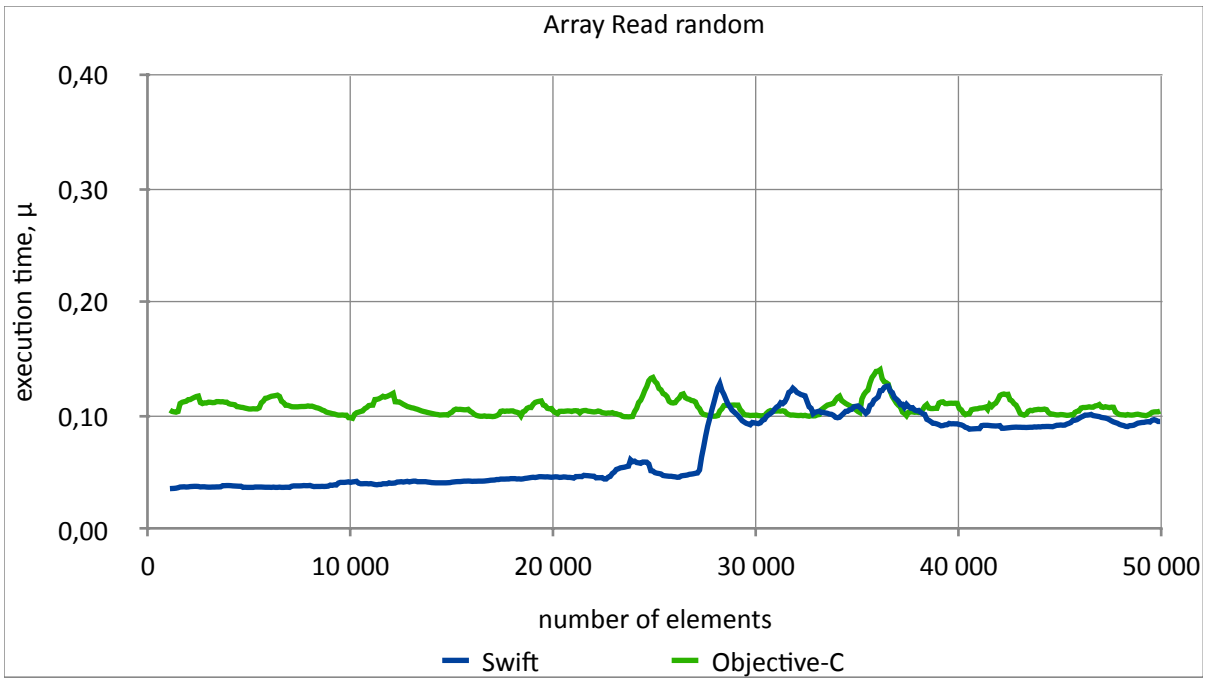
В целом видно что Swift 3-4 раза быстрее Objective-C и время исполнения более стабильно.



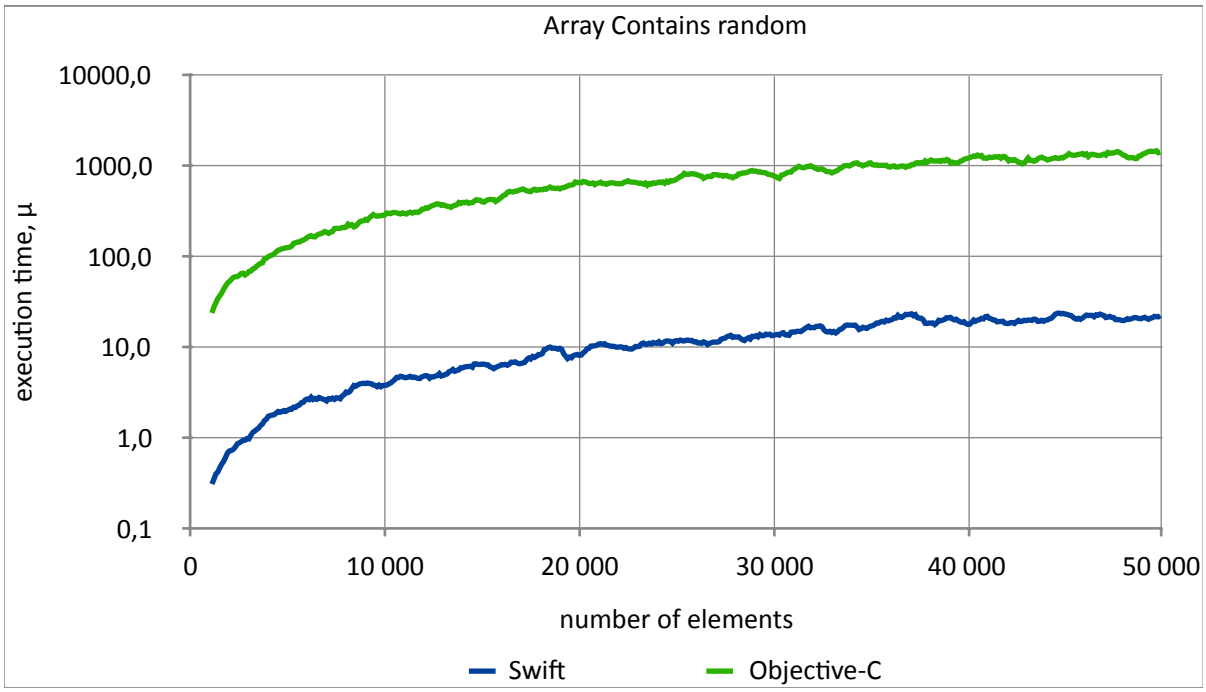

Здесь оба языка ведут себя примерно одинаково.



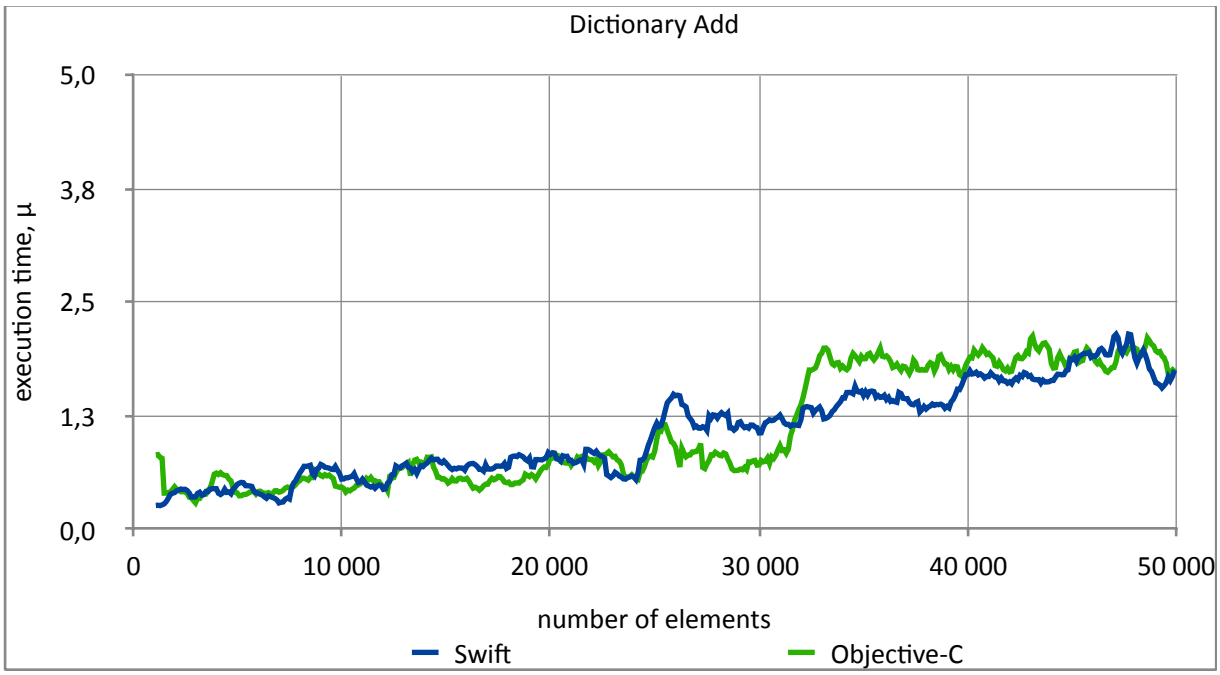

Время удаления элемента массива линейно зависит от его размера у обоих языков, но явно видно, что Objective-C примерно в 2 раза быстрее.



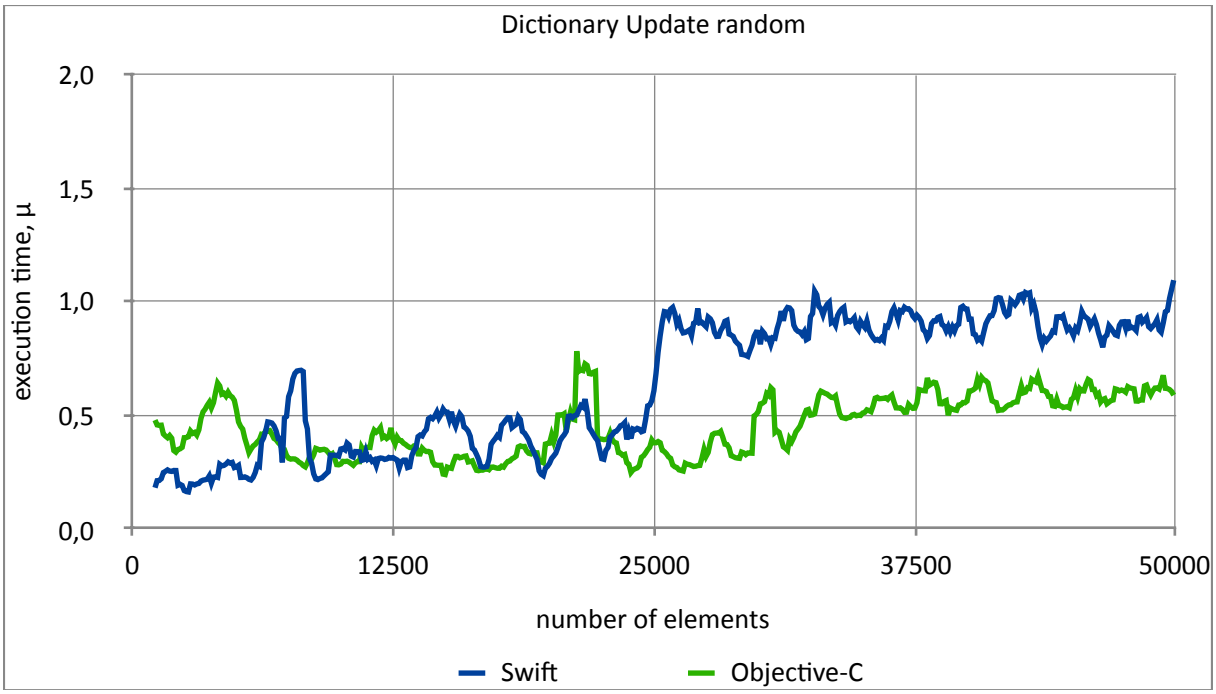

При малых размерах массива Swift в несколько раз быстрее, но примерно начиная с 27 000 элементов, времена мало отличаются, хотя Swift остается более быстрым.



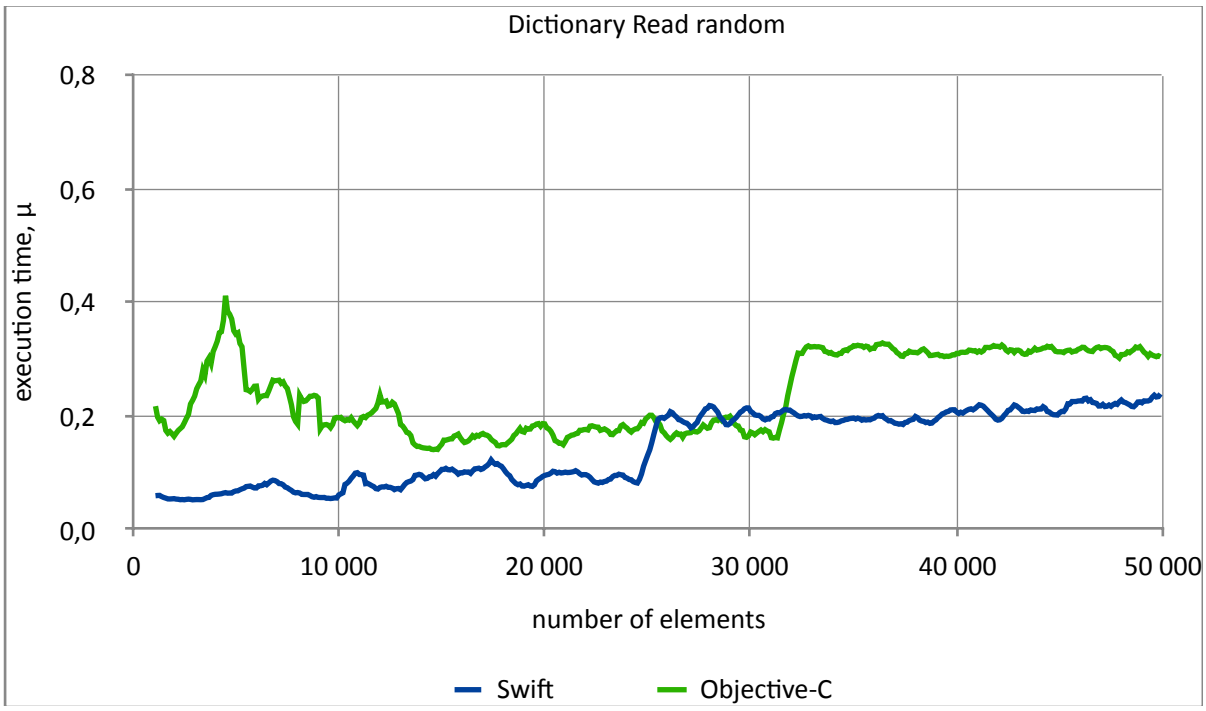

В этом тесте Swift оправдал свое название, и в 60-80 раз обогнал своего соперника, время проверки линейно зависит от размера массива.



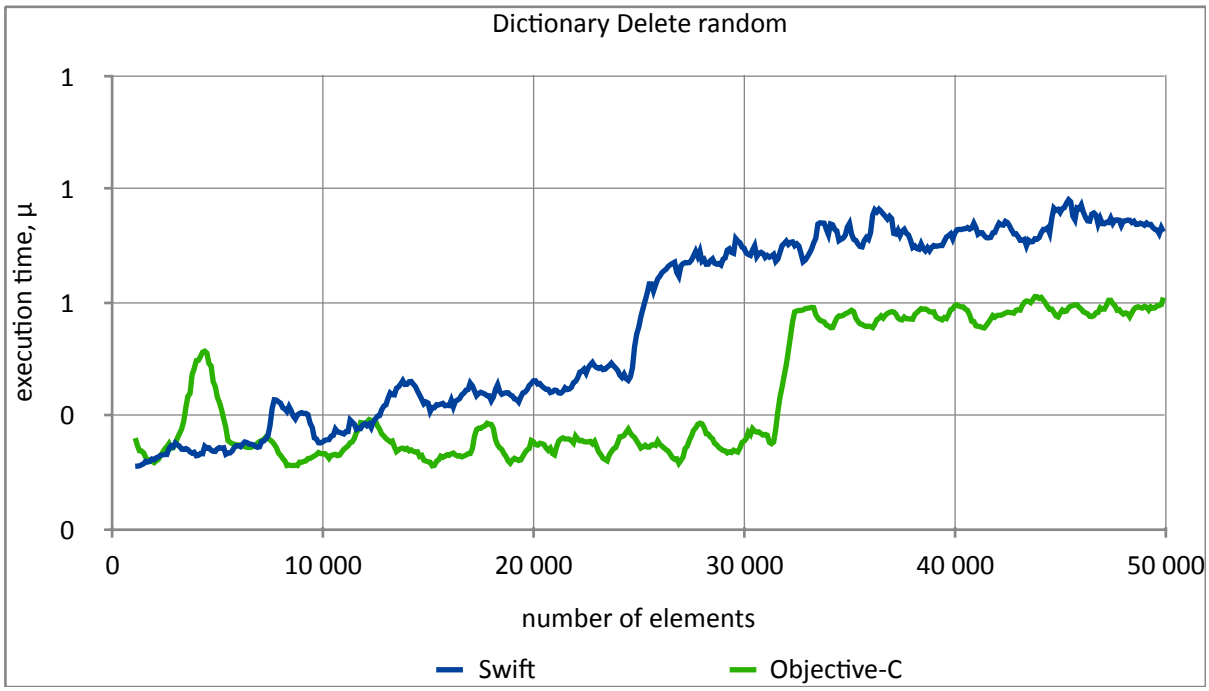

И для Swift и для Objective-C ведут себя примерно одинаково, хорошо видно увеличение накладных расходов на операцию с ростом словаря.



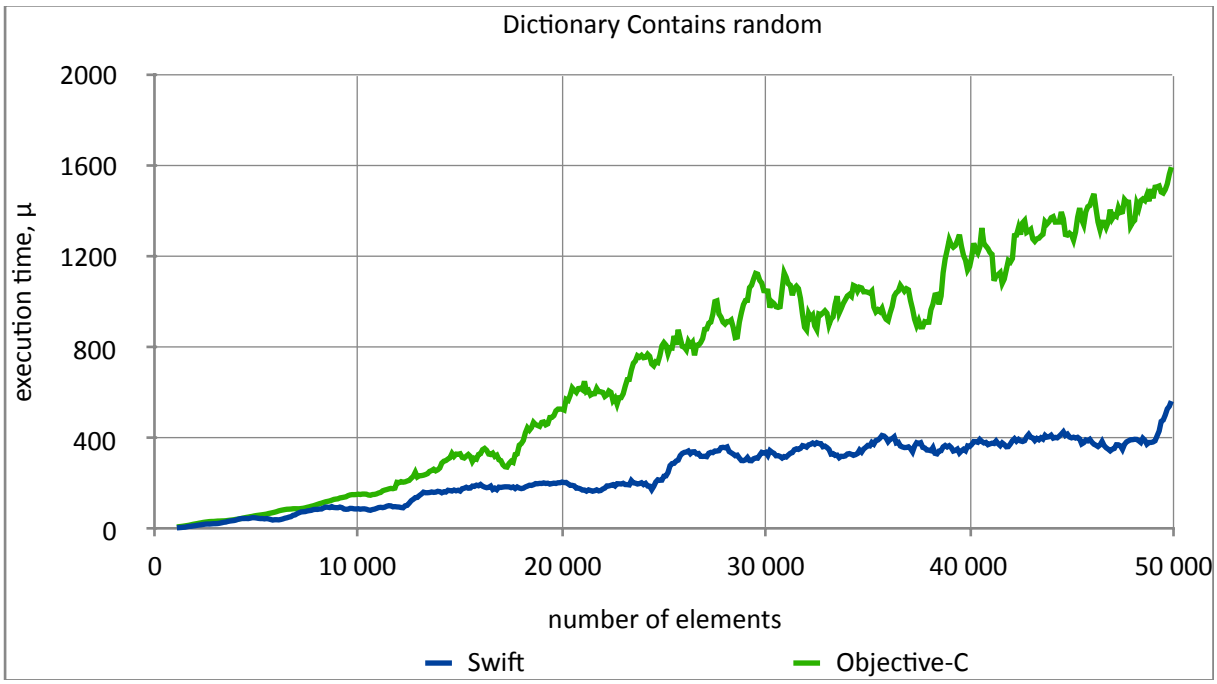

При небольших размерах словаря времена совпадают в пределах погрешности, но начиная с 25 000 у Swift увеличиваются накладные расходы.



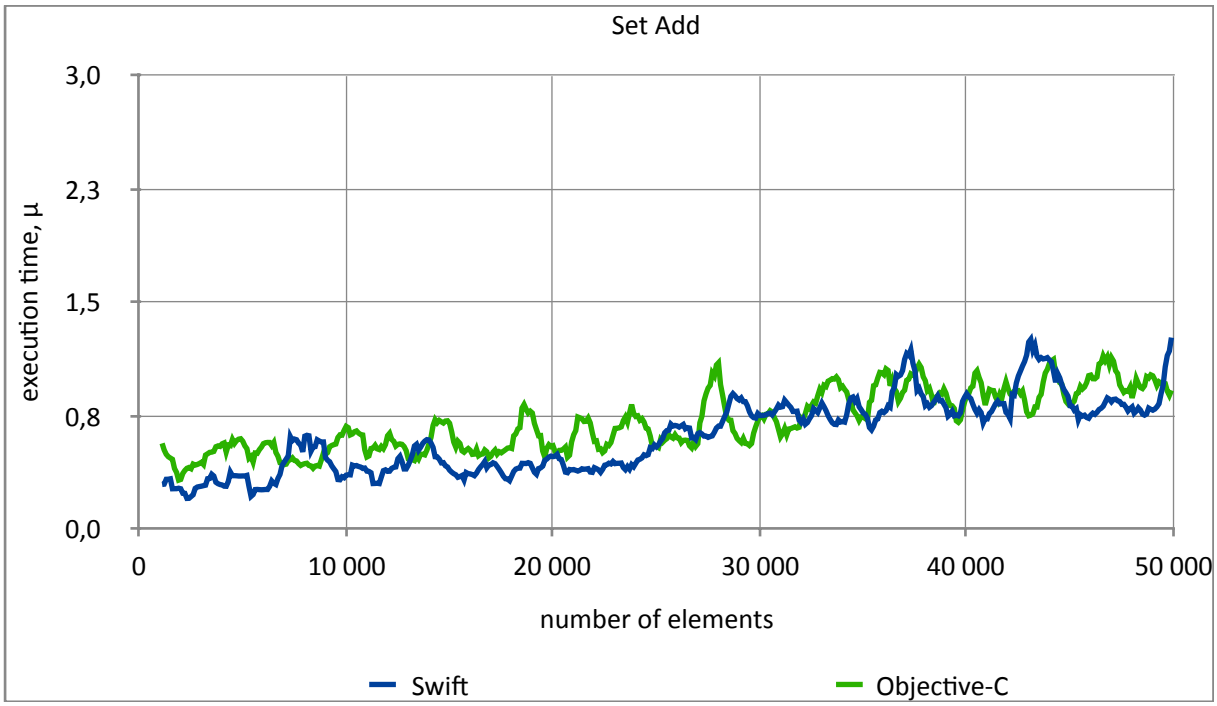

Видно что Swift немного лучше чем Objective-C, и время операции почти не зависит от размера словаря.



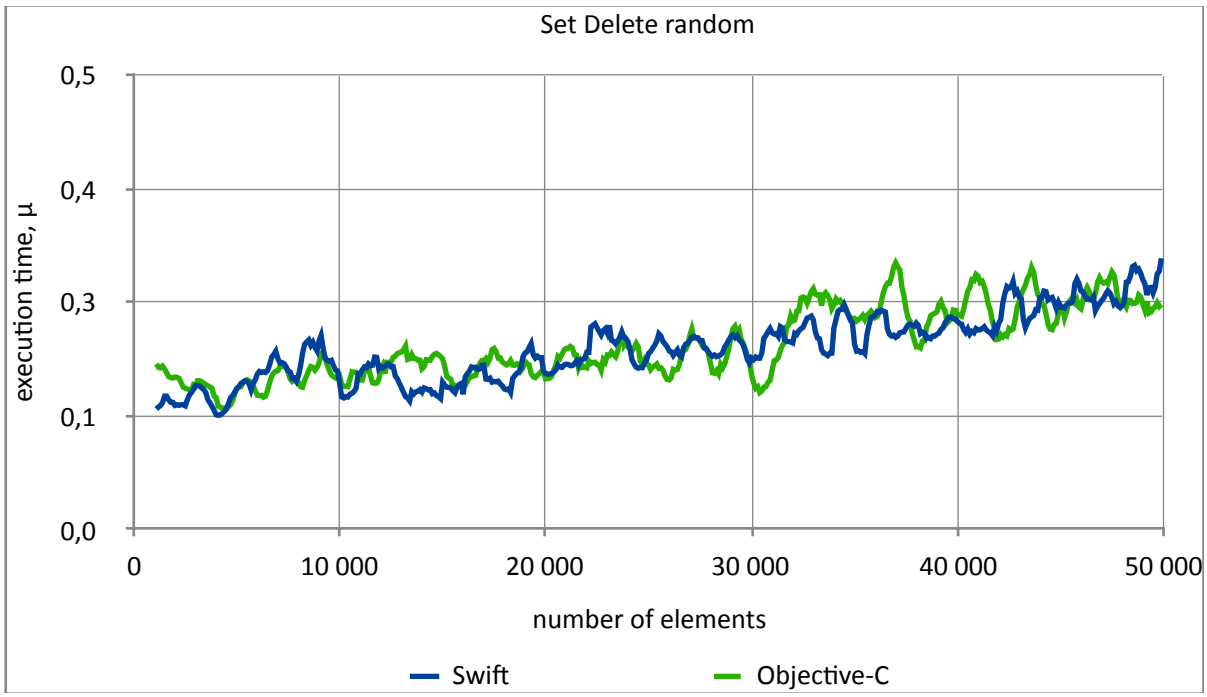

В удалении Objective-C все-таки выигрывает, снова начиная с 25 000 элементов.



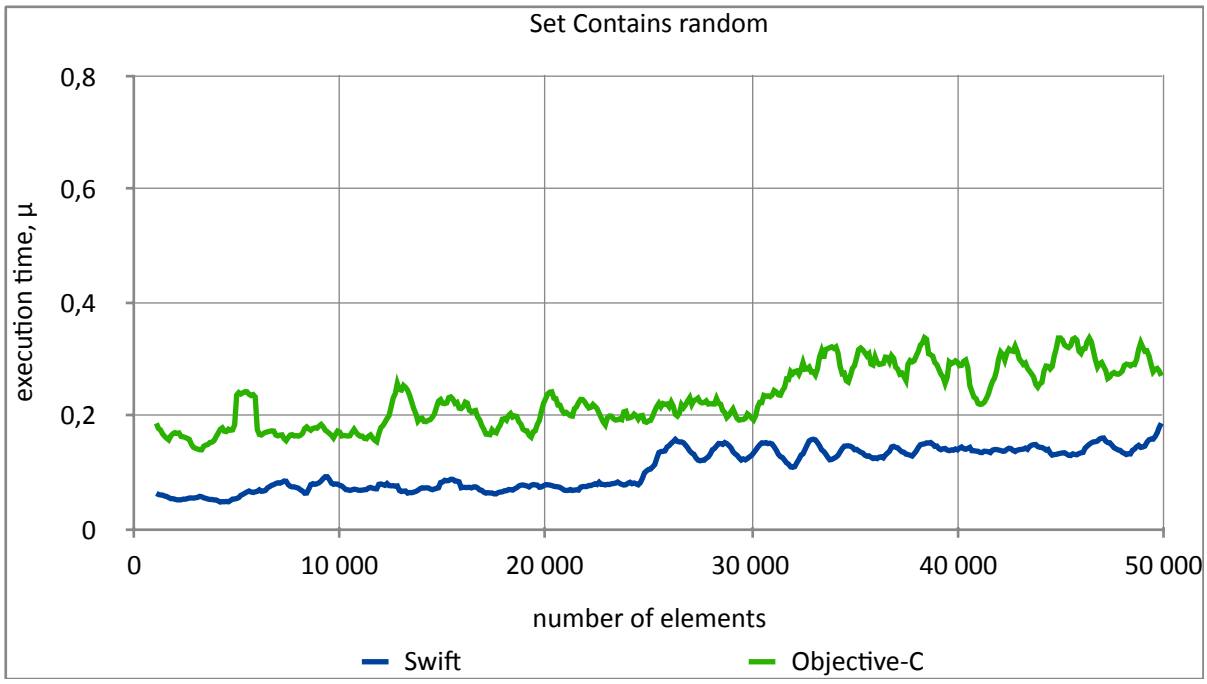

В проверке что данный элемент присутствует в словаре время операции линейно зависит от размера словаря, и у Objective-C оно доходит до 1500 микро секунд. Swift быстрее в 2-3 раза.




Время добавления элемента не зависит от размера множества (реализуется через хеш-таблицу), и одинаково для обоих языков.




Время удаления тоже не зависит от размера множества, и тоже одинаково у обоих языков.




Время проверки присутствия элемента тоже не зависит от размера множества, но Swift немного быстрее.

### 6.4 Случаи медленной компиляции нескольких строчек языка Swift

При всех своих достоинствах Swift обладает еще одним недостатком - большое время компиляции. При разработке это также очень важный фактор. В более ранних версиях языка было довольно много случаев, когда компилятор думал над некоторыми строчками по пол секунды, вместо положенных десятков миллисекунд. В Swift 3 с этим стало лучше - большинство подобных примеров стали компилироваться за приемлемое время, но некоторые все еще актуальны. Вот некоторые из них:

```
// 12690.0 ms
func getSize1() -> CGSize {
   return CGSize(width: size.width + (rightView?.bounds.width ?? 0) +
        (leftView?.bounds.width ?? 0) + 22, height: bounds.height)
}
// 37.9 ms
func getSize2() -> CGSize {
   var padding: CGFloat = 22
   if let rightView = rightView {
      padding += rightView.bounds.width
   }

   if let leftView = leftView {
      padding += leftView.bounds.width
   }
   return CGSize(width: size.width + padding, height: bounds.height)
}
```

Можно сделать вывод, что нужно избегать использования оператора ?? в сложных конструкциях и заменять его на if let.

Здесь функция возвращает разные массивы строк в зависимости от type.

```
// 377.2 ms
func typeNames1(type: Int) -> [String] {
   let labelNames = type == 0 ? (1...5).map{String($0)} : (0...2).map{String($0)}
   return labelNames
}
```

```
// 12.2 ms
func typeNames2(type: Int) -> [String] {
    var labelNames: [String]
    if type == 0 {
        labelNames = (1...5).map{String($0)}
    } else {
        labelNames = (0...2).map{String($0)}
    }
    return labelNames
}
```

Отсюда следует что в сложных выражениях использование тернарного оператора сильно замедляет компиляцию.

Также если определять тип возвращаемого значения, с использованием AnyObject, где возвращаемое значение - константа например трижды вложенный словарь, записанная без явного указания типа, то компилятор тратит много времени на его определение. В преложении пример 1 демонстрирует нам что время компиляции уменьшается в 13 раз при явном указании типа.

И конечно, не стоит писать длинные логические выражения, если они хорошо ложатся на switch. В приложении в примере 2 показано, что при замене такого выражения на switch время компиляции уменьшилось в 9 раз.

# 7. Результаты

1) По результатам измерения времени сортировки видно, что при включении оптимизации Swift становится заметно быстрее чем Objective-C, в некоторых случаях даже в десятки раз. А стандартная сортировка даже обгоняет qsort() из C, что можно объяснить тем, что qsort() - универсальный метод, использующий вашу функцию сравнения, а Swift пользуется тем что массив состоит из Int.
2) В полиморфизме, если нужен динамический полиморфизм, то при создании массива, наследование и протокольные типы ведут себя одинаково, но при проходе по массиву наследование выигрывает в 2,5 раза если структура не помещается в 3 слова (буфер контейнера) и проигрывает в 1,6 раз если помещается. Если можно использовать статический полиморфизм, то лучше всего шаблоны с опцией Whole Module Optimization. При создании они быстрее на 30%, а при проходе по массиву обгоняют конкурентов в 6-9 раз.
3) Измерение времени операций коллекций элементов
    1. Array
        a. add - swift в 2 быстрее
        b. update - одинаково
        c. delete - Objective-C в 2 быстрее
        d. read - на небольших размерах массива swift быстрее
        e. contains - Swift в десятки раз быстрее
    2. Dictionary
        a. add - одинаково
        b. update - одинаково
        c. read - swift немного быстрее
        d. delete - Objective-C немного быстрее
        e. contains - swift в 2 раза быстрее
    3. Set
        a. add - одинаково

b. delete - одинаково

c. contains - Swift в 2 раза быстрее

В заключение можно сказать, что коллекции элементов в Swift работают в целом быстрее, проигрывает только удаление элементов. Но в Objective-C постоянно работает динамический выбор реализации метода, в то время как Swift дает возможность использовать эту удобную, но замедляющую программу возможность, только при необходимости. Это позволяет писать на Swift действительно быстрые программы.

# 8. Дальнейшее развитие

В дальнейшем нужно внимательно рассмотреть инструменты, которые относятся к самому языку - это стандартная библиотека и Foundation.

Стандартная библиотека - это набор функций которые доступны без импорта каких-либо библиотек, например Foundation. Этот набор включает в себя 74 функции, 7 из них документированы в книге по Swift «The Swift Programming Language», остальные относятся к стандартным протоколам. Вот функции которые не относятся к стандартным протоколам -

1) assert(true)
2) countElements("foo")
3) for (i, j) in enumerate(["A", "B"]) { … }
4) min(8, 3, 5)
5) print("Hello ")
6) println("World")
7) sort(["B", "A"])

Фреймворк Foundation обеспечивает основной уровень функциональности, необходимый для работы приложений. Его разделяют на несколько частей.

1. **Основные**
   - 1.a. Числа, данные и основные типы
   - 1.b. Строки и текст
   - 1.c. Коллекции элементов
   - 1.d. Дата и время
   - 1.e. Вычисление размерных величин
   - 1.f. Преобразование чисел, дат, измерений и других значений в представления строк
   - 1.g. Фильтрация и сортировка
2. **Поддержка приложений**
   - 2.a. Управление задачами
   - 2.b. Управление ресурсами

2.c. Уведомления

2.d. Взаимодействие с расширением приложения.

2.e. Ошибки и исключения

2.f. Поддержка скриптов

**3. Файлы и сохранение данных**

3.a. Файловая система

3.b. Архивы и специализация

3.c. Информация о настройках приложения

3.d. Поиск

3.e. Облачное хранилище

**4. Сеть**

4.a. Система загрузки URL

4.b. Поиск устройств в локальной сети

**5. Низко уровненные утилиты**

5.a. Безопасное межпроцессорное взаимодействие

5.b. низкоуровневая поддержка основных особенностей Objective-C

5.c. Процессы и треды

5.d. Ввод, вывод между файлами, процессами, сетями (stream, pipe, socket, port)

Из всего этого множества нужно будет выбрать те инструменты, которые сильно влияют на производительность и энергопотребление типичных приложений, например взять несколько известных приложений определить в каких функциях они тратят больше всего времени. И написать для этих инструментов тесты. Также можно написать приложение, которое запускаясь на iPhone показывает сравнительные характеристики Swift и Objective-C.

# 9. Приложение

### Пример 1

```
// 133 ms - до
class myClass1 {
    let myCompany: Dictionary<String, AnyObject>
    init() {
        let myCompany = [
            "employees": [
                "employee 1": ["attribute": "value"],
                "employee 2": ["attribute": "value"],
                "employee 3": ["attribute": "value"],
                "employee 4": ["attribute": "value"],
                "employee 5": ["attribute": "value"],
                "employee 6": ["attribute": "value"],
                "employee 7": ["attribute": "value"],
                "employee 8": ["attribute": "value"],
                "employee 9": ["attribute": "value"],
                "employee 10": ["attribute": "value"],
                "employee 11": ["attribute": "value"],
                "employee 12": ["attribute": "value"],
                "employee 13": ["attribute": "value"],
                "employee 14": ["attribute": "value"],
                "employee 15": ["attribute": "value"],
                "employee 16": ["attribute": "value"],
                "employee 17": ["attribute": "value"],
                "employee 18": ["attribute": "value"],
                "employee 19": ["attribute": "value"],
                "employee 20": ["attribute": "value"],
            ]
        ]
        self.myCompany = myCompany as Dictionary<String, AnyObject>
    }
}
```

```
// 10.1 ms - после
class myClass2 {
    let myCompany: Dictionary<String, AnyObject>
    init() {
        let myCompany: Dictionary<String, Dictionary<String, Dictionary<String,
String>>> = [
            "employees": [
                "employee 1": ["attribute": "value"],
                "employee 2": ["attribute": "value"],
                "employee 3": ["attribute": "value"],
                "employee 4": ["attribute": "value"],
                "employee 5": ["attribute": "value"],
                "employee 6": ["attribute": "value"],
                "employee 7": ["attribute": "value"],
                "employee 8": ["attribute": "value"],
                "employee 9": ["attribute": "value"],
                "employee 10": ["attribute": "value"],
                "employee 11": ["attribute": "value"],
                "employee 12": ["attribute": "value"],
                "employee 13": ["attribute": "value"],
                "employee 14": ["attribute": "value"],
                "employee 15": ["attribute": "value"],
                "employee 16": ["attribute": "value"],
                "employee 17": ["attribute": "value"],
                "employee 18": ["attribute": "value"],
                "employee 19": ["attribute": "value"],
                "employee 20": ["attribute": "value"],
            ]
        ]
        self.myCompany = myCompany as Dictionary<String, AnyObject>
    }
}
```

**Пример 2**

```
// 62.7 ms - до
enum Food1 {
    case beef
    case broccoli
    case chicken
    case greenPepper
    case lettuce
    case onion
    case redPepper
    case spinach
    case artichoke
    case cabbage
    case celery
    case kale
    case radish
    case squash
    case parsley
    case yellowPepper

    func isVegetable1() -> Bool {
        return self == .broccoli ||
            self == .greenPepper ||
            self == .lettuce ||
            self == .onion ||
            self == .redPepper ||
            self == .spinach ||
            self == .artichoke ||
            self == .cabbage ||
            self == .celery ||
            self == .kale ||
            self == .radish ||
            self == .squash ||
            self == .parsley ||
            self == .yellowPepper
    }
}
```

```
// 0.7 ms - после
enum Food2 {
    case beef
    case broccoli
    case chicken
    case greenPepper
    case lettuce
    case onion
    case redPepper
    case spinach
    case artichoke
    case cabbage
    case celery
    case kale
    case radish
    case squash
    case parsley
    case yellowPepper

    func isVegetable2() -> Bool {
        switch self {
            case .broccoli, .greenPepper, .lettuce,
                .onion, .redPepper, .spinach,
                .artichoke, .cabbage, .celery,
                .kale, .radish, .squash,
                .parsley, .yellowPepper:
            return true
        case .beef, .chicken:
            return false
        }
    }
}
```

# 10. Список литературы


1) https://yalantis.com/blog/is-swift-faster-than-objective-c/
2) https://github.com/apple/swift
3) https://medium.com/@RobertGummesson/regarding-swift-build-time-optimizations-fc92cdd91e31
4) https://developer.apple.com/documentation
5) https://appleinsider.ru/ios/10-prichin-vyuchit-swift-i-razrabatyvat-prilozheniya-dlya-ios.html
6) iOS and macOS Performance Tuning: Cocoa, Cocoa Touch, Objective-C, and Swift (Developer's Library) - Marcel Weiher
7) http://gadgets-news.ru/swift-vs-objective-c-vs-si/
8) http://perfectorium.com/objective-c-ili-swift
9) http://swiftme.ru/89-2/
10) https://habrahabr.ru/post/308450/
11) https://habrahabr.ru/post/107126/
12) http://swiftbook.ru
13) https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/CollectionTypes.html
14) https://habrahabr.ru/post/198104/
15) https://habrahabr.ru/company/kingservers/blog/307012/
16) http://itmozg.ru/news/1286/#.WVFD2cZeO9Y
17) http://irace.me/swift-profiling
18) https://appleinsider.ru/tag/swift
19) https://ru.wikipedia.org/wiki/Worldwide_Developers_Conference
20) https://developer.apple.com/videos/play/wwdc2016/416/
21) https://developer.apple.com/videos/play/wwdc2015/409/
22) https://chriswong1.gitbooks.io/wwdc-2015/content/Optimizing%20Swift%20Performance.html
23) https://lifehacker.ru/2014/07/22/kak-swift-navsegda-izmenit-mir-programmirovaniya/

24) https://dou.ua/lenta/articles/swift-vs-objective-c/
25) https://dev.by/lenta/main/yazyk-swift-krupneyshiy-proryv-apple
26) https://web-academy.com.ua/stati/14-stati/65-zachem-uchit-objective-c-esli-apple-sozdala-yazyk-programmirovaniya-swift