**Плюсы Swift:**

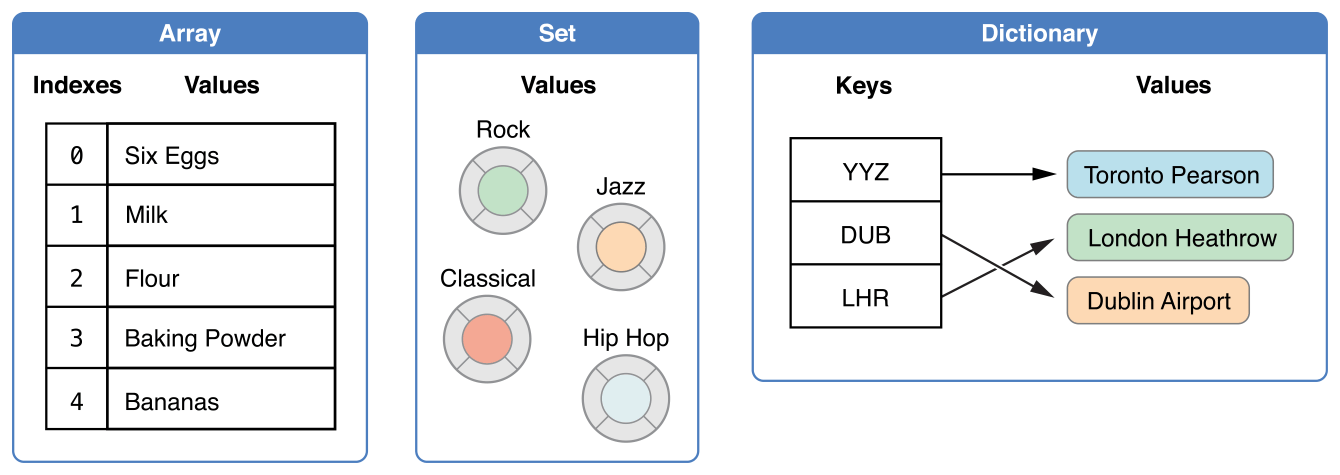
* Строгая типизация;
* Встроенная обработка ошибок;
* Современные синтаксические конструкции: замыкания, дженерики, кортежи;
* Безопасность: защита от переполнения, Optional`ы, которые позволяют больше не бояться nil;
* Отличная читаемость кода;
* Хорошая документация.

**Что такое компилятор**

Компиляцией называют преобразование исходного кода в машинный (или в другое низкоуровневое представление), который можно потом использовать для создания исполняемого файла.

**Коллекции:**

Коллекции бывают изменяемые и неизменяемые, если var то это изменяемая коллекция (можем дописывать какие-то данные во внутрь или изменять имеющиеся). Если это let то коллекцию мы изменять не можем, разрешено только чтение.



**Множества (Set) - Value Type**

* **Н**еупорядоченная коллекция (всегда разный порядок в print)
* Уникальные элементы в отличии от массива (пишу 1,2,3,3 будет 1,2,3)
* Тоже имеет индекс как и массив
* Может содержать только один тип данных

Для объявления set нужно это писать явно:



Чтобы упорядочить set можно вызвать метод sorted. В этом случае set вседа будет выводится в print упорядоченно.

* **Поиск** - O(1) в среднем случае (поиск по хеш-числу)
* **Добавление** - O(1) в среднем случае (поиск по хеш-числу)
* **Удаление** - O(1) в среднем случае (поиск по хеш-числу)

В худшем случае если будут коллизии то O(N) для всех операций.

**Коллизия** - это когда совпадают хеш-коды у элементов.

**Как устроен?**

Внутри скрытый массив, где индексом будет хеш-код (числовое значение), полученный через хеш-функцию, а значение кладется в массив (все фундаментальные типы Hashable).

Если элементы разные, то кладутся в ту же ячейку памяти, но как связный список (Linked List).

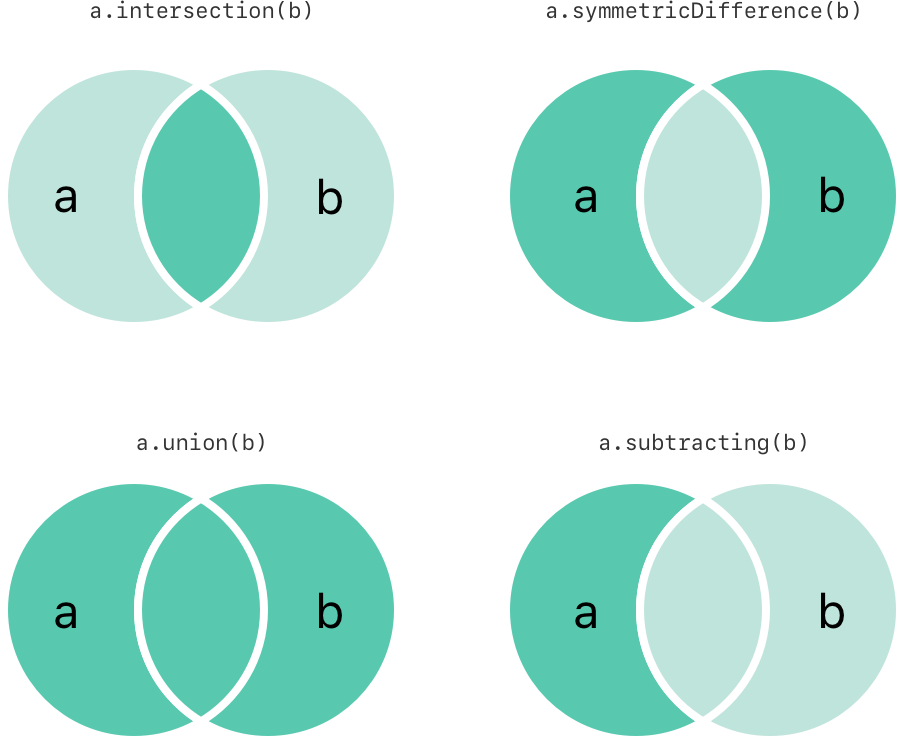
Если хеш-код и значение одинаковы, то элемент не кладется, так как уже лежит ячейке.

Работают с Hash значением из-за этого быстрее чем массив. Ключ должен соответствовать протоколу Hashable. (Int, String, Double, Bool) являются хешируемыми типами по умолчанию и могут быть использованы в качестве типов значений множества или в качестве типов ключей словаря. Множества используют хеш-таблицу для значений.

Элемент → Хеш-функция → Хеш-код → Индекс массива

Для массивов и множеств большая разница заключается в механизме поиска элементов в коллекции. Массив должен проверять каждый элемент один за другим. Множество может напрямую обращаться к значению на основе хэша, без необходимости проверять всю коллекцию.

**Основные операции**

****

* intersection(\_:) метод для создания нового набора, содержащего только значения, общие для обоих наборов.
* symmetricDifference(\_:) создает set из несовпадающих элементов.
* union(\_:) объединяет.
* subtracting(\_:) метод для создания нового набора со значениями, не входящими в указанный набор.

**Словари (Dictionary) - Value Type**

[Ключ] : [Значение]. Работает быстрее чем массив из-за hash. Неупорядоченные значения. Работают с Hash значением. Ключи должны быть подписаны на протокол Hahsable. Ключ словаря может быть целым числом или строкой, но он должен быть уникальным. Словари используют хэш-таблицу для ключей. Передает пару ключ-значение в хэш-таблицу для хранения.

**Это контейнер хранящий в себе несколько значений одного и того же типа. Каждое значение связано с уникальным ключом, который выступает в качестве идентификатора этого значения внутри словаря. Не упорядоченная коллекция.** [Key] : [Value]



**Как устроен** dictionary (Hash Map)**?**

Тот же самый механизм, что и у Set. Отличие только в том, что хеш-код берется не от элемента, а от ключа и кладется в массив не элемента, а как пара key: value

Ключ → Хеш-функция → Хеш-код → Индекс массива

**Массив (Array) - Value Type**

* Не уникальные значения (элементы повторяются)
* Упорядоченная коллекция элементов одного и того же типа (пишу 1,2,3,4 будет в print 1,2,3,4 и причем к Int нельзя добавить String)
* Может хранить другие массивы
* Размер массива нам известен
* Зная индекс можно очень быстро получить нужное значение
* Чтение - O(1) (чтение по индексу)
* Поиск - O(N) (проходим по всему массиву и ищем подходящий элемент)
* Добавление - O(1) в лучшем случае и O(N) в худшем случае

- если в начало и надо сдвигать массив

- если в конец и надо переносить непомещающийся массив в новую память

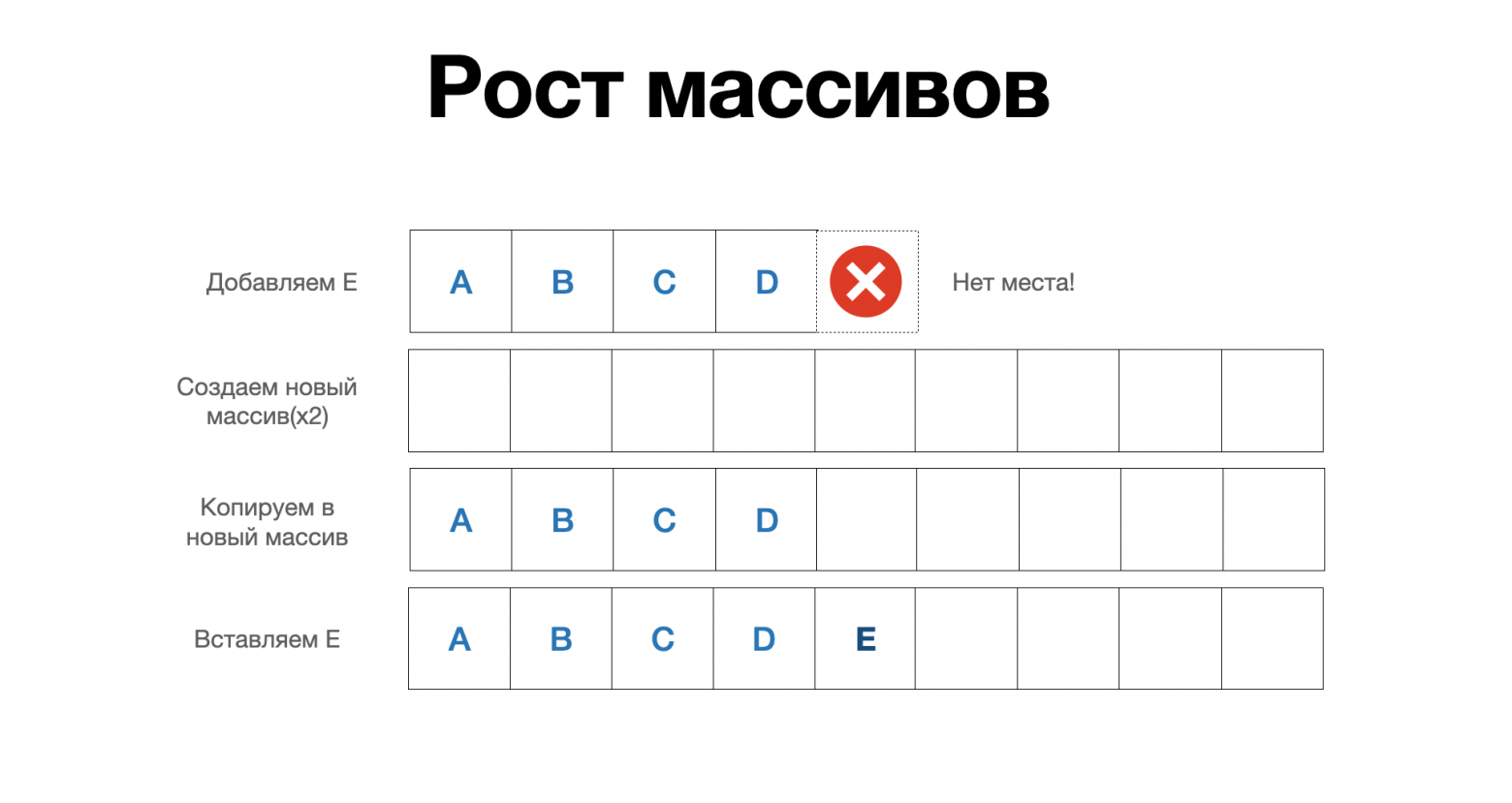
* Удаление - O(N) (при удалении надо сдвигать элементы)

Как устроен?

Выбирается сплошной участок памяти, делится на части по индексам c 0, и упорядоченно кладутся все элементы. Для массивов и множеств большая разница заключается в механизме поиска элементов в коллекции. Массив должен проверять каждый элемент один за другим. Множество может напрямую обращаться к значению на основе хэша, без необходимости проверять всю коллекцию.

Как хранится в памяти?

Элементы массива хранятся упорядоченно в памяти, где доступом к участку памяти будет индекс. Рост массива ниже.

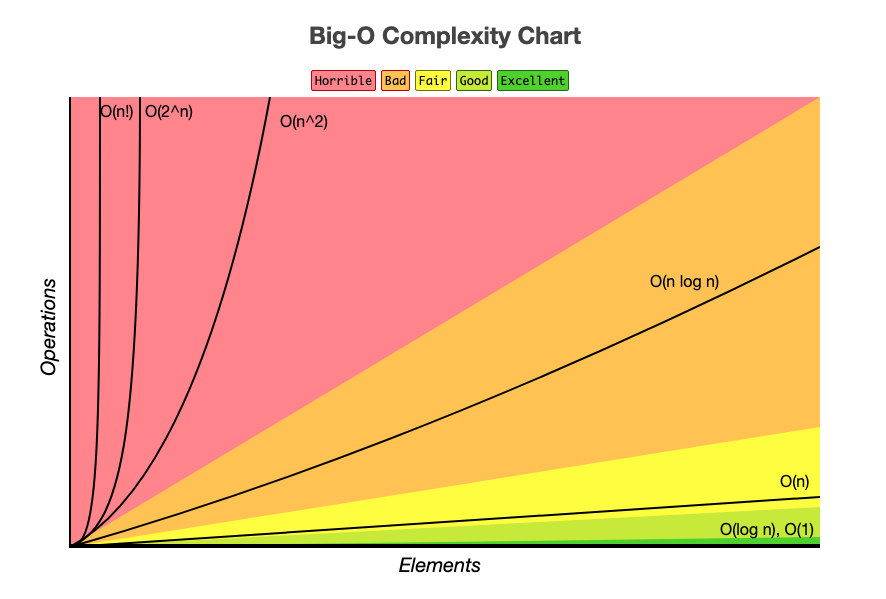


Изначально массив запрашивает в оперативной памяти определенный объем пространства для хранения собственных элементов. Максимально доступное количество элементов хранится в свойстве capacity. Как только значение свойства count (текущее количество элементов) превысит capacity - происходит реаллокация, то есть массив перемещается в новую область памяти. При этом значение capacity растет экспоненциально, всегда увеличиваясь в 2 раза. Нужно это чтобы не снижать производительность при каждом изменении размера массива.

Несмотря на интеллектуальное распределение памяти, наиболее эффективный способ выделения памяти – это если вы знаете емкость, для которой должен быть определен массив. Таким образом, требуется только одно выделение памяти. Массивы Swift предоставляют возможность определять и резервировать емкость "на лету", и это может быть сделано с небольшим приростом производительности. Через метод –

# **Big O**

Способ оценки относительной производительности структуры данных или алгоритма, обычно по двум осям: времени и пространству.



# **O (1) постоянное время – приведенный выше код получает элемент из массива, используя индексный номер. Неважно, содержит ли этот массив 10 элементов или 10.000, для получения элемента массива по его индексному номеру всегда потребуется одно и то же время. Сложность поиска значения в массиве равна O(1). Почему это происходит так быстро? Алгоритм извлечения элемента из массива по номеру индекса может напрямую вычислять адрес памяти этих данных на основе индекса. Ему не нужно выполнять итерации или поиск по массиву – это прямая, мгновенная операция.**

O (n) линейное время – означает, что время, необходимое алгоритму для завершения, растет линейно с размером входных данных. При удвоенных затратах нам нужно выполнить вдвое больший объем работы.

O (log n) логарифмическое время – такие алгоритмы, как Двоичные Деревья Поиска (Бинарные Деревья Поиска), очень быстры, потому что они половинят свои результаты каждый раз, когда ищут результат. Это деление пополам является логарифмическим, которое мы обозначаем как "O(log n)".

O (n2) квадратичное время – когда вы встраиваете один цикл for-in в другой, вы получаете квадратичный эффект, применяемый к вашему алгоритму, который может сильно замедлить работу. Это нормально для получения правильного ответа, просто они не самые производительные.

**Уровни доступа**

Уровни доступа определяют доступность объектов и методов. Если объект закрыт уровнем доступа, то по ошибке обратиться к нему не получится, он просто не будет доступен. Конечно, можно игнорировать уровни доступа, но это снизит безопасность кода.

* Public - Обычно его используют для фреймворков. Модули имеют доступ к публичным объектам других модулей. Наследоваться с ним не можем.
* Internal - Внутренний уровень стоит по умолчанию для свойств и методов. Явно указывать internal не требуется.

Эти записи равнозначны:

var number = 3

internal var number = 3

* Fileprivate – Этот уровень доступа позволяет использовать объект в пределах его исходного файла. Используйте файл-частный уровень доступа для того, чтобы спрятать детали реализации определенной части функциональности, когда эти части функциональности будут использоваться внутри другого файла.
* Private - Ограничивает доступ к свойствам и методам внутри структур, классов и перечислений. private — самый строгий уровень, он скрывает вспомогательную логику.
* Open - Похож на public - разрешает доступ из других модулей. Используется только для классов, их свойств и методов. Наследоваться с ним можем.

В чем разница между уровнем доступа Fileprivate и Private?

Fileprivate – на этом уровне расположен доступ к элементам данных и функциям текущего файла. Используется для скрытия реализации, требуемой только в текущем исходном файле. Private – самый низкий уровень доступа. Ограничивает использование сущности, которая включается декларацией или расширением в текущем файле. При этом доступ в подклассах или в других файлах отсутствует. Private – разрешить доступ к членам данных и функциям в рамках их объявления или расширения в текущем файле.  Fileprivate – разрешить доступ к членам данных и функциям в одном и том же исходном файле или в подклассе, или расширении.

**Дженерики (Generics) и Any**

Универсальные шаблоны, которые разрешают создавать универсальные функции и типы. Работают с каждым типом в соответствии с требованиями, которые определяет разработчик.

Главная особенность – пишется один код, который не дублируется для использования с другими типами.

Разница между any и джинерик – джинерики более безопасны

Протоколы для работы с дженериками:

1. Numeric – позволяет плюсовать / вычитать / умножать / делить свойства
2. Equatable – проверяет на равенство

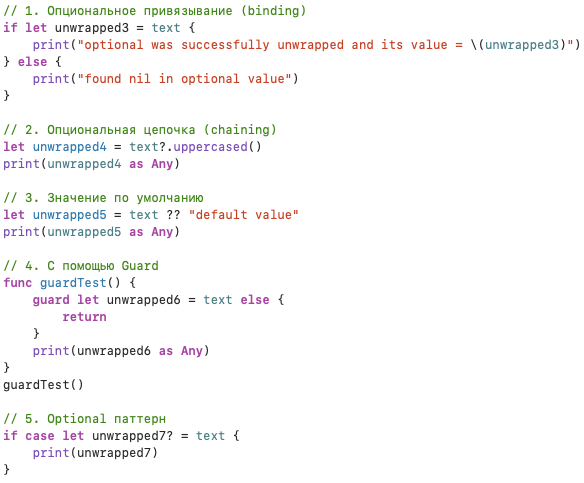
* Any – используется для всех типов
* AnyObject – используется для типов Class

**Optional (Generic)**

Optional – тип, который используется в Swift каждый раз при работе с необязательным значением (может быть а может нет). Это обвертка над остальными типами (Float, Double, Int, String и т.д). Это enum которые который имеет два значения: some (дженерик, куда помещается какой-то тип данных) и none = nil, сам по себе Optional не может существовать, может только с каким-то типом данных.

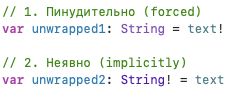
nil – обозначает отсутствие значения какого-либо типа.

Безопасные развертывания Optional:



Не безопасные развертывания Optional

Если при обращение к переменной которая была не безопасно развернута и в ней пустота (nil) то будет краш;



Можно использовать только когда мы 100% уверены, что будет значение. В идеале сразу делать безопасное развертывание при создании сомнительных типов данных.

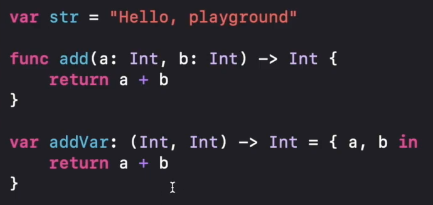
Unwrapped – развернутое значение

**Замыкания (closures) Reference Type**

Кложур = замыкания = лямбда функции – необходимы чтобы упростить синтаксис в нашей программе, сделать запись более легкой, без названий параметров и т.д.

Кложуры и функции это почти одно и тоже, только у кложура нет имени. Их можно передавать как параметр в другие функции.

Чаще всего под кложурами подразумевается безымянные замыкания.

Пример записи функции и кложура

Свифт понимает что если в функции или в кложуре всего одна строчка, и эта функция что-то возвращает, то слово return можно не писать. Запись будет выглядеть вот так:



Но и тут можно сократить кложур, заменив имена параметров на цифры



Бывают: Escaping и Non-escaping closures

* Escaping – могут быть сохранены и использованы в будущем
* Non-escaping – выполняют код немедленно и не сохраняют его

Completion handlers

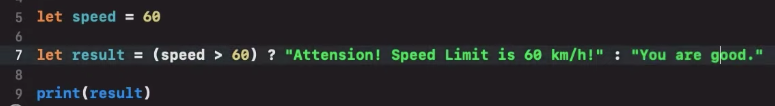
Completion handlers (обработчики завершения) — это замыкания в действии. Предположим, вы выполняете трудоемкую задачу, например сетевой запрос, и хотите что-то сделать сразу после завершения запроса.

Но вы определенно не хотите тратить ресурсы впустую, проверяя несколько раз, продолжается ли процесс или нет. Здесь используются обработчики завершения. Обработчик завершения — это замыкание, которое «вернется» сразу после завершения трудоемкого процесса.

**Тернарный оператор**

Является важным оператором ветвления. Нужен чтобы сократить код.

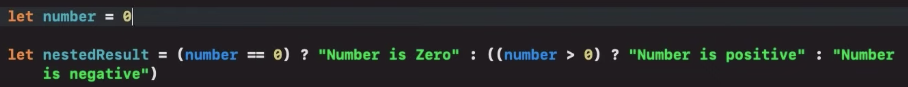
Условия? первое условие (если условие выполняется): второе условие(если условие не выполняется)



В print будет You are good, потому что условие false

решение такого же примера, но уже с if else

Вложенный тернарный оператор

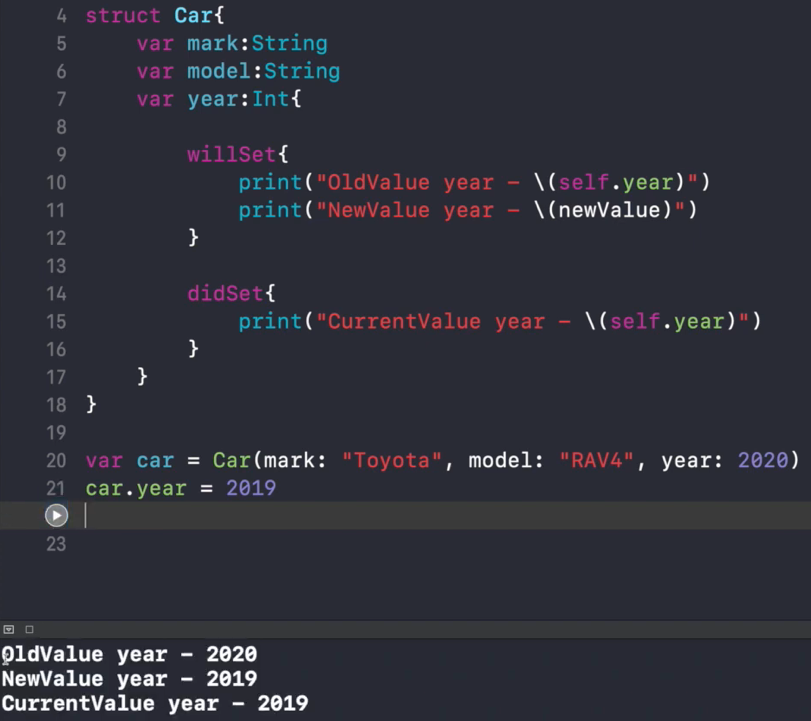


**Перечисление (enum) Value Type**

Название с большой буквы. В Enum нельзя объявлять т.е хранить в себе данные Stored Properties – это отличие от структуры и класса, можно только Computed Properties. По умолчанию инициализатор отсутствует. Создает новый тип данных в нашем приложении. Чтобы переопределить enum нужно как в структурах использовать mutating перед func. В enum можно делать все то, что можно делать в class и struct.

**Properties (свойства)**

Рroperty observers (наблюдатели) следят за изменениями свойств – willSet (будет установлено прямо сейчас), didSet (было установлено только, что). Эти два метода вызываются каждый раз когда значения текущего Properties меняется. Они не вызываются во время инициализации, работают только после изменения свойства после создания экземпляра (важно знать).



Пока car не поменяли на 2019 в консоли OldValue, newValue и CurrentValue не было.

Getter – чтение данных. По умолчанию используется с return, если не используется set то return можно не писать.

Setter – назначение нового значения

У willset – есть параметр newValue

У didset – есть параметр oldValue

Вычисляемые (computed properties) – не хранят в себе значения а лишь вычисляют их

Хранимые свойства (stored properties) – контейнер где хранятся данные

Lazy properties – применимы в основном в Stored Properties и подписаны как lazy var, let не может быть lazy. Инициализируется (появляется в памяти) лишь тогда когда к этому свойству обращаются напрямую.

(Свойства типа) Properties Type – Static – говорит что константу можно использовать напрямую на типе данных, не используя для этого объекты. Пример работы при выборе цвета, пишем UIKit.black в этом случае мы не создаем этот объект а сразу работаем напрямую с типом.

Работает также как Lazy т.е пока мы к нему не обратимся их нет. В статик можно работать еще с функциями – в этом случае функцию можно вызвать прямиком на типе и не создавать объект в памяти чтобы не засорять ее. Еще можно работать не только со Static но еще и с class func – в этом случае и статик и класс технически работают одинаково но добавляет больше возможностей связанных с наследованием в классах, т.е эту функцию можно переопределить в классе наследнике со статиком такой возможности не будет. В первую очередь это влияет на наследования.

Self. – это свойство которое хранит в себе ссылку на сам экземпляр.

**Протоколы (protocols) это value type**

Это абстрактный тип данных. Абстрактный – это значит не существует явных экземпляров у данного типа, существуют только экземпляры которые могут соответствовать этому типу (быть подписанными под этот тип). Служит для объединения структур, классов или других типов данных, т.е эти типы данных должны соответствовать определенному набору правил. Нужен для чистоты архитектуры.

Работают с class, struct, enum.

Если создаются несколько протоколов они могут наследоваться друг от друга. Если мы наследуем протоколы значит мы наследуем требования других протоколов.

Протокол работает с (get и set) – чтение(получение) и запись.

Если мы подписываем под протокол структуру у нее не может быть weak проперти потому что weak бывают только у классов. Чтобы избежать этой ошибки нужно указать у протокола явно что он для класса. protocol Primer: class { }

**Hashable - протокол**

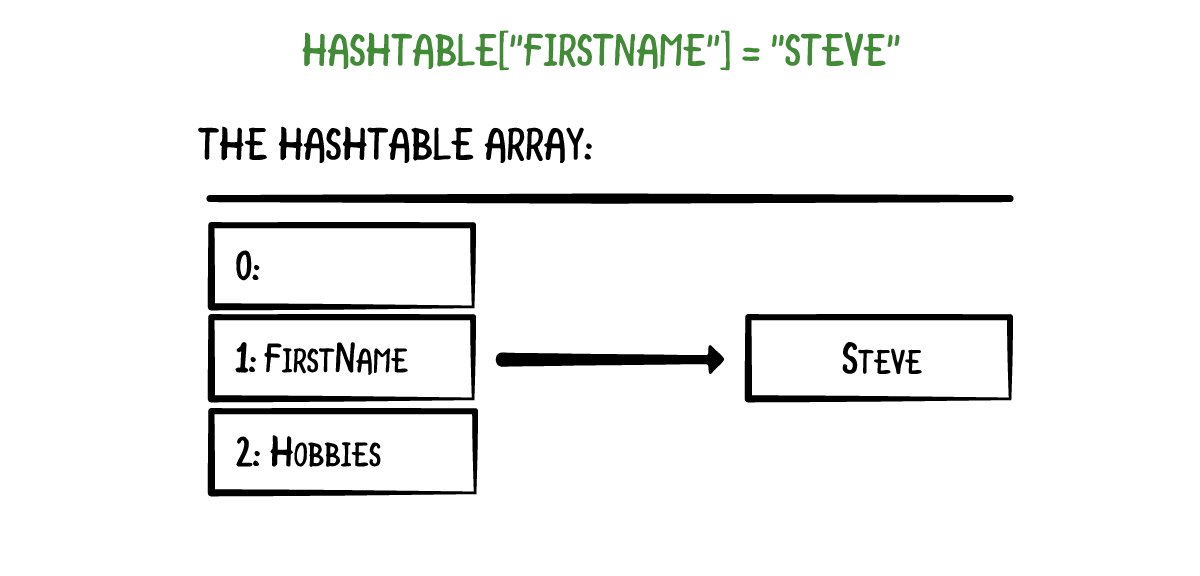
Одна из причин, по которой так часто используются хэш-таблицы, заключается в том, что они очень эффективны. Временные сложности для поиска в хэш-таблице, вставки и удаления элементов составляют в среднем O (1). Это означает, что время работы остается постоянным независимо от размера входных данных.

Предоставляет хэш-значения (hash value) экземплярам. К хранимым свойствам в массиве мы обращаемся по индексу, с dict или set мы ищем значения по hash value т.е по значению hash. Некоторые операции в сете и словаре работают быстрее чем в массиве

HashValue – это цифровое значение, оно непостоянное (398548754375243957)

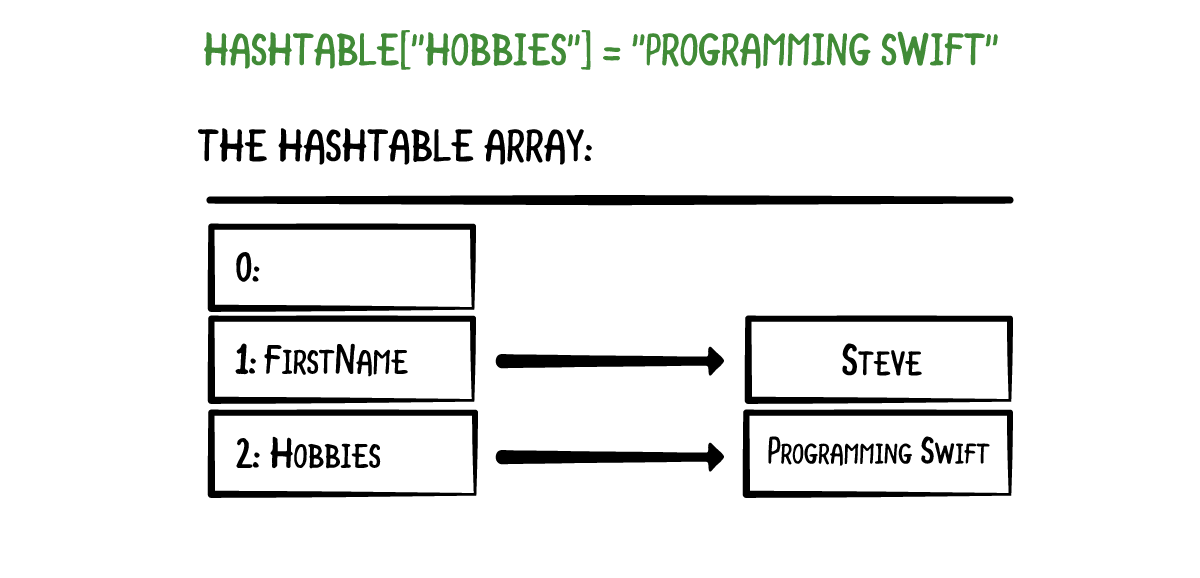
Если HashValue совпадают это называется коллизия

Хэш-таблица - это массив. Изначально он пуст. Когда вы помещаете значение в хэш-таблицу под определенным ключом, он использует этот ключ для вычисления индекса в массиве. Позволяет хранить и извлекать объекты по ключу.



В этом примере ключ "firstName"соответствует индексу массива 1.

Добавление значения под другим ключом помещает его в другой индекс массива:



Хитрость заключается в том, как хэш-таблица вычисляет эти индексы массива. Вот тут-то и возникает хеширование. Когда вы пишете следующее утверждение,



хэш-таблица принимает ключ “FirstName” и запрашивает у него его hashValue свойство. Следовательно, ключи должны соответствовать Hashable протоколу.

Что общего между классами и структурами:

● Они могут определять свойства и методы.

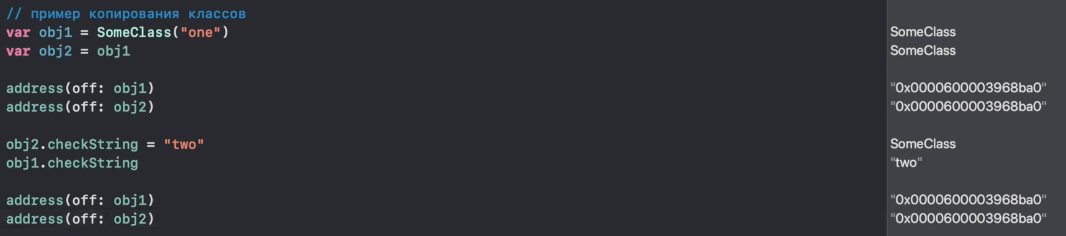
● Они могут определять сабскрипты.

● Они могут определить инициализаторы с помощью функции init().

● Они могут использовать расширения.

● Они могут соответствовать протоколам.

**Класс (Reference Type – ссылочный тип)**

Ссылочные типы, с другой стороны, хранят указатель на адрес памяти данных. Это означает, что каждый новый экземпляр, назначенный этой ссылке, будет указывать на один и тот же адрес, тем самым совместно используя свою копию данных.

Нужны для того чтобы объединять данные и функционал.

Создает новый тип данных в нашем приложении

Как запретить наследование класса? Сделать класс конечным, используя ключевое слово final

Лежит в отдельном виде памяти (heap – куча) на эту память ссылается какое-то количество ссылок, когда мы ссылку передаем в функцию или присваиваем в переменную она просто копируется таких ссылок на этот объект становится больше но объект в памяти лежит один

Все что создается внутри класса, структуры или перечисления (var name и т.д) – это свойства класса или объекта или т.д называются Properties (проперти).

В классе могут храниться вычисляемые (computed properties) и хранимые свойства(stored properties).

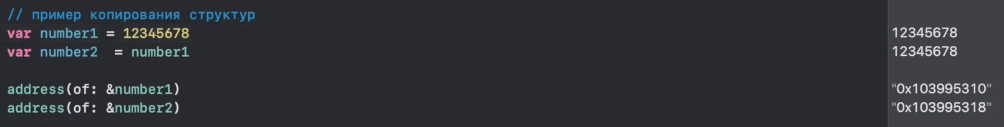
Чем отличается класс от объекта (класса)?

Класс это просто описание того как данные и функции будут лежать в памяти или как мы будем с ними взаимодействовать, оно не занимает никакой памяти. Когда создается готовый объект это то что занимает объект в памяти, это что-то что имеет конкретные значения.

Потокобезопасность – к ссылочным типам можно получить доступ практически из любого потока в любое время. Если 2 потока обращаются к объекту одновременно могут возникать ошибки.

**Структура (Value Type – значимый тип)**

Тип значения сохраняет уникальную копию своих данных. Это означает, что если вы присвоите его другой переменной, будет создана новая копия данных, и обе переменные будут работать независимо.



Создает новый тип данных в нашем приложении

Структуры не могут наследоваться друг от друга потому что они value type, но могут быть подписаны под протоколы.

В структуре могут храниться вычисляемые(computed properties) и хранимые свойства (stored properties).

Mutating – разрешает изменения свойств структуры в методе. Работает только в структурах или enum

Когда лучше выбрать структуру:

* Если не хотим использовать наследование то лучше структуру;
* Не хотим тратить ресурс на работу ARC;
* Инкапсулируется малый набор данных;
* Значение копируется, а не передается по ссылке;
* Потокобезопасность - поскольку структуры являются уникальными копиями, в большинстве случаев управлять ими одновременно может только один поток.

Отличие классов от структур:

* Класс Reference type, Сктруктура Value type;
* Структуры работают быстрее;
* Class – Heap, Struct – Stack;
* В структуре есть встроенный инициализатор в классах нет;
* В классах есть наследование в структурах нет;
* У структуры нет деинициализатора;
* В классах есть оператор идентичности === **он** проверяет, указывают ли два объекта на один и тот же объект в памяти. Это полностью отличается от оператора **==**, который проверяет, равны ли два значения;
* Структуры потокобезопастны а класс нет.

**Расширение (extension)**

Помогают расширить функциональность для: классов, структур и enum. С их помощью можно добавить новый метод, новое вычисляемое свойство, инициализатор и т.д.

**Поведенческий паттерн Delegate**

* Делегатор - тот, кто ставит задачу;
* Тип делегата - протокол, которому должен соответствовать делегат;
* Делегат - тот, кто берет на себя обязанность выполнять задачу.

**Delegate и Data Source:**

**Delegate**

Нужен чтобы оповещать об изменениях в таблице.

**Data Source**

Механизм чтобы запрашивать какие-то данные из вне с которыми этот класс работает. Нужно чтобы таблица понимала сколько у нее ячеек.

# **NotificationCenter**

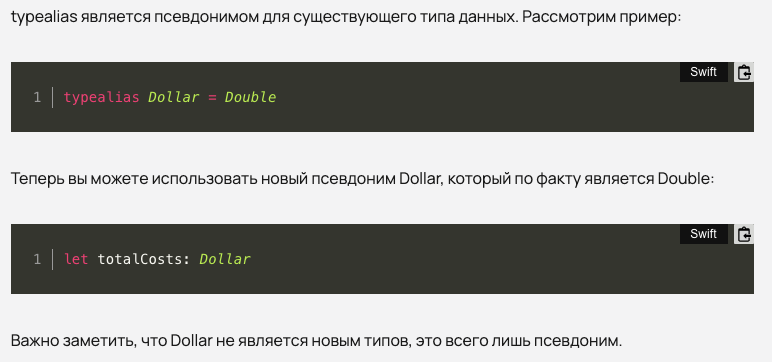
Позволяет прослушивать события и реагировать на них. Apple описывает его как “механизм отправки уведомлений, который позволяет передавать информацию зарегистрированным наблюдателям. Нужен для передачи данных из одного класса или структуры в другой класс или структуру, вызов любых других методов или функций. Можно воспользоваться шаблоном делегирования который по сути лучше, но делегирование не поможет для одновременной трансляции одиночных сообщений или уведомления методов. Объекты которые прослушивают эти события называются наблюдателями.

* Это помогает для наблюдения и трансляции синглов из одного класса в несколько классов;
* Отправляйте одиночные сообщения из одного класса в несколько классов;
* Получение синглов из нескольких классов.

**Оператор вида «??»**

Он возвращает значение в левой части, если оно не nil. Если левая часть равна nil, то возвращается значение в правой части.

**Typealias**



**Defer**

Означает работу, которую мы хотим сделать при любых обстоятельствах. Это может быть абсолютно любая работа: метод, закрывающий файл или 100 строк кода, которые нужны для чистки. Очень важно: Swift гарантирует, что эта работа будет выполнена до того, как закончится ее область видимости.

defer — это ключевое свойство, которое обеспечивает выполнение блока кода, когда остальная текущая область не выполняется

Одной из мощных особенностей defer является то, что можно назначать несколько отложенных кусков кода к исполнению и Swift гарантирует их выполнение. Более того, Swift исполняет отложенные куски кода в обратной последовательности. То есть, то, что вы отложили в последнюю очередь, исполнится первым, и наоборот. Таким образом, получается обратный стек.

Это означает, что вам не стоит беспокоиться об исполнении отложенных работ, а также, что стек defer исполняется именно в том порядке, в котором вы его создали.