Типы данных в Objective-C

Objective-C является надмножеством C (это значит, что любая программа написанная на C будет являться программой на Objective-C), из этого следует, что все типы данных из C будут присутствовать в Objective-C. То есть нам доступны как примитивные типы int, float, double char, указатели, так и C-массивы, структуры, перечисления, объединения. Кроме этого в Objective-C добавляются свои псевдонимы над примитивами и собственные объектные типы для работы со строками NSString, с числами NSNumber, с датами NSDate, с кодом как с объектами (это блоки, но сейчас мы их рассматривать не будем). Подробное рассмотрение типов языка C в данном курсе не предусмотрено, мы ожидаем, что слушатель понимает, что такое указатели, массивы в C и знает некоторые подходы по работе с данными из C.

Примитивы C

Тип Void

Void в переводе пустота. Функция возвращающая тип void не возвращает ничего.

```
void helloWorld() {
    NSLog(@" Hello world! Данная функция ничего не вернет.»);
}
```

Не стоит путать тип void с указателем void *.

Целочисленные типы

Целочисленные типы данных характеризуются их длиной и наличием знака (signed и unsigned). Тип char всегда занимает 1 байт, но нужно очень хорошо запомнить, что конкретные размеры целочисленных типов *зависят от реализации*. Гарантируется только что short <= int <= long <= long long. Примеры:

```
BOOL isBool = YES;
NSLog(@"%d", isBool);
```

```
NSLog(@"%@", isBool ? @"YES" : @"NO");

char aChar = 'a';
unsigned char anUnsignedChar = 255;
NSLog(@"The letter %c is ASCII number %hhd", aChar, aChar);
NSLog(@"%hhu", anUnsignedChar);

short aShort = -32768;
unsigned short anUnsignedShort = 65535;
NSLog(@"%hd", aShort);
NSLog(@"%hu", anUnsignedShort);

int anInt = -2147483648;
unsigned int anUnsignedInt = 4294967295;
NSLog(@"%d", anInt);
NSLog(@"%u", anUnsignedInt);

long aLong = -9223372036854775808;
unsigned long anUnsignedLong = 18446744073709551615;
NSLog(@"%ld", aLong);
NSLog(@"%lu", anUnsignedLong);

long long aLongLong = -9223372036854775808;
unsigned long long anUnsignedLongLong = 18446744073709551615;
NSLog(@"%lld", aLongLong);
NSLog(@"%llu", anUnsignedLongLong);
```

Модификаторы %d и %u отвечают за вывод целых со знаком и без. hh, h, l, ll в NSLog отвечают за вывод char, short, long и long long сооветственно.

BOOL является целым и является частью Objective-C, а не C. В качестве значений BOOL используются YES и NO.

Целые фиксированной длины

Так как фактический размер целочисленных зависит от реализации, то для некоторых случаев более удобно выбрать тип с определенной длиной.

ПРИМЕР

```
// Exact integer types
int8_t aOneByteInt = 127;
uint8_t aOneByteUnsignedInt = 255;
int16_t aTwoByteInt = 32767;
uint16_t aTwoByteUnsignedInt = 65535;
int32_t aFourByteInt = 2147483647;
uint32_t aFourByteUnsignedInt = 4294967295;
int64_t anEightByteInt = 9223372036854775807;
uint64_t anEightByteUnsignedInt = 18446744073709551615;

// Minimum integer types
int_least8_t aTinyInt = 127;
uint_least8_t aTinyUnsignedInt = 255;
int_least16_t aMediumInt = 32767;
uint_least16_t aMediumUnsignedInt = 65535;
int_least32_t aNormalInt = 2147483647;
uint_least32_t aNormalUnsignedInt = 4294967295;
int_least64_t aBigInt = 9223372036854775807;
uint_least64_t aBigUnsignedInt = 18446744073709551615;

// The largest supported integer type
intmax_t theBiggestInt = 9223372036854775807;
uintmax_t theBiggestUnsignedInt = 18446744073709551615;
```

Числа с плавающей точкой

В C присутствуют три типа для чисел с плавающей точкой. Картина с ними такая же как и с целочисленными в C, их размер тоже зависит от реализации, дается гарантия что float <= double <= long double.

В коде литералы представляющие float должны заканчиваться на f, а long double на L.

ПРИМЕР

```
// Single precision floating-point
float aFloat = -21.09f;
NSLog(@"%f", aFloat);
NSLog(@"%8.2f", aFloat);
```

```
// Double precision floating-point
double aDouble = -21.09;
NSLog(@"%8.2f", aDouble);
NSLog(@"%e", aDouble);

// Extended precision floating-point
long double aLongDouble = -21.09e8L;
NSLog(@"%Lf", aLongDouble);
NSLog(@"%Le", aLongDouble);
```

Определение размера для типа

Для определения размера используется функция sizeof(), которая возвращает количество байт, которое тип занимает в памяти. Использование данной функции является самым простым способом посмотреть какой размер имеет тот или иной зависимый от реализации тип на конкретной архитектуре.

ПРИМЕР

```
NSLog(@"Size of char: %zu", sizeof(char));   // This will always be 1
NSLog(@"Size of short: %zu", sizeof(short));
NSLog(@"Size of int: %zu", sizeof(int));
NSLog(@"Size of long: %zu", sizeof(long));
NSLog(@"Size of long long: %zu", sizeof(long long));
NSLog(@"Size of float: %zu", sizeof(float));
NSLog(@"Size of double: %zu", sizeof(double));
NSLog(@"Size of size_t: %zu", sizeof(size_t));
```

При использовании sizeof() на массиве вернется количество байт занимаем массивом. Встает вопрос, а какое количество элементов можно хранить в массиве? Тут на помощь приходит тип size_t:

```
size_t numberOfElements = sizeof(anArray)/sizeof(anArray[0]);
```

Аспекты работы с примитивами C

Сейчас будут рассмотрены различные неочевидные аспекты работы с примитивами C.

Выбор целых

Так как в C довольно много вариантов целочисленных типов, не всегда понятно какой из них выбрать. Совет простой - используйте int всегда, когда у вас нет резона использовать какой другой тип. Однако стоит сделать одну оговорку. Для индекса массивов стоит использовать целые без знака (индексы никогда не бывают отрицательные).
int традиционно близок к нативному машинному слову текущей архитектуры.

Деление целых

Если в операции деления оба аргументы целые, то будет использоваться целочисленная арифметика, однако если хотя бы один аргумент будет с плавающей точкой, то будет использоваться арифметика с плавающей точкой, с неявным преобразованием типа аргумента. Это стоит помнить.

ПРИМЕР:

```
int integerResult = 5 / 4;
NSLog(@"Integer division: %d", integerResult);        // 1
double doubleResult = 5.0 / 4;
NSLog(@"Floating-point division: %f", doubleResult);  // 1.25
```

Сравнение чисел с плавающей точкой на равенство

Необходимо запомнить, что числа с плавающей точкой нельзя сравнивать на равенство.

ПРИМЕР:

```
NSLog(@"%.17f", .1);        // 0.10000000000000001
```

Из-за особенностей представления чисел с плавающей точкой на реальном железе результат операции 1.0 - 0.9 никогда не будет тождественно равен 0.1 .

Подобные сравнения безопасно делать используя либо проверку на бесконечно малую (сравнения разности чисел с плавающей точкой с наперед заданной константой, если разность меньше нее, то числа равны), либо для определенных применений, например хранения и

выполнения операций над денежными суммами, используйте NSDecimalNumber.

Примитивы Objective-C

К имеющимся примитивам C добавляются примитивы Objective-C: id, SEL, Class, которые являются базисом динамической природы Objecive-C, также кратко рассмотрим псевдонимы NSInteger и NSUInteger.

id

id это общий тип для всех объектов в Objective-C, для простоты понимания его можно сравнивать с указателем void * из C. И также как и void * id может хранить ссылку на любой объект.

ПРИМЕР:

```
id mysteryObject = @"An NSString object";
NSLog(@"%@", [mysteryObject description]);
mysteryObject = @{@"model": @"Ford", @"year": @1967};
NSLog(@"%@", [mysteryObject description]);
```

Class

Класс представляет собой сам класс объекта. Воспользовавшись им можно например проверить динамически проверить тип объекта:

```
Class targetClass = [NSString class];
id mysteryObject = @"An NSString object";
if ([mysteryObject isKindOfClass:targetClass]) {
    NSLog(@"Yup! That's an instance of the target class");
}
```

SEL

Тип SEL - внутреннее представление имени метода. Манипуляции с SEL используются для различных операций связанных с динамизмом языка. В примере ниже имя метода сохраняется в переменную:

```
SEL someMethod = @selector(sayHello);
```

NSInterger, NSUInteger

Код написанный на Objective-C работает на разных типах архитектур: 32-битных, 64-битных. Из-за разного размера C-типов (чуть раньше мы

говорили что их размер зависит от реализации) писать код под разные архитектуры не всегда удобно и не всегда приятно (можно поймать весьма забавные сайд эффекты). Поэтому для упрощения перехода между архитектурами были сделаны NSInteger и NSUInteger, которые дают единый размер целочисленной переменной на разных архитектурах.

ОПРЕДЕЛЕНИЯ ИЗ NSOBJCRUNTIME.H

```
#if __LP64__ || (TARGET_OS_EMBEDDED && !TARGET_OS_IPHONE) ||
TARGET_OS_WIN32 || NS_BUILD_32_LIKE_64
typedef long NSInteger;
typedef unsigned long NSUInteger;
#else
typedef int NSInteger;
typedef unsigned int NSUInteger;
#endif
```

Домашнее задание

Освежить указатели, прочитать про перечисления (enum), посмотреть на сайте Apple документацию на NSString, NSNumber, NSDecimalNumber.

Коллекции в Objective-C

Также как и в других развитых языках программирования, в Objective-C присутствуют все необходимые виды коллекций: массивы, словари, множества и их различные вариации. Кратко рассмотрим синтаксис и общие особенности, а после вернемся к подробному рассмотрению каждой из коллекций.

СОЗДАНИЕ И ОБХОД МАССИВА

```
NSArray *animals = @[@"Cat",@"Dog", @"Bird"];
for (NSString *animal in animals) {
     NSLog(@"%@", animal);
}
```

СОЗДАНИЕ СЛОВАРЯ И ОБХОД ПО КЛЮЧАМ

```
NSDictionary *legsAndPaws = @{
                                    @"Cat":@(4),
```

```
                                    @"Dog":@(4),
                                   @"Bird":@(2),
                                 };
for (NSString *animal in legsAndPaws.allKeys) {
     NSLog(@"%@ has %@ legs (or paws).", animal, legsAndPaws[animal]);
}
```

Изменяемость и неизменяемость

В Objective-C для изменяемых типов в имени будет присутствовать часть «Mutable», например *NSString, NSMutableString, NSDictionary, NSMutableDictionary, NSArray, NSMutableArray* и т.п. Это означает, что однажды созданный объект нельзя модифицировать. На первый взгляд может показаться, что неизменяемые коллекции неудобны для применения, в них нельзя менять состав элементов, менять элементы местами, удалять их. Однако существует несколько классов задач и применений в которых подобная неизменяемость является преимуществом, например работа с несколькими потоками и передачи между ними объектов.

Какие типы можно хранить

Во всех коллекциях Objective-C можно хранить только объектные типы. Для скалярных типов, например *BOOL, int, double* придется воспользоваться контейнером *NSNumber*. Также не получится в коллекции сохранить *nil*, если нужно сохранить в коллекции что-то с семантикой пустого объекта придется воспользоваться специальным объектом *NSNull*.

Коллекции и управление памятью

NSArray, NSDictionary, NSSet их варианты имеют одно и то же поведение с точки зрения управления памятью - они удерживают объект при добавлении в коллекцию (увеличивают счетчик ссылок на него). В NSDictionary ключи имеют дополнительную особенность, но на ней стоит подробней остановиться при рассмотрении словарей.

Дженерики

В развитых языках программирования существуют параметризированные коллекции, в которых в виде параметра выступает тип - дженерики.

C#. СПИСОК СО ЗНАЧЕНИЯМИ ТИПА INT

```
List<int> listA = new List<int>();
```

В современном Objective-C также появились языковые конструкции, которые Apple назвала «легковесные дженерики» (lightweight generics).

OBJECTIVE-C. ДЖЕНЕРИКИ. ПРИМЕР СИНТАКСИСА

```
NSArray<NSString *> *animals = @[@"Cat",@"Dog", @«Bird"]; //(1)
NSDictionary<NSString *, NSNumber *> *legsAndPaws = @{
                                                    @"Cat":@(4),
                                                    @"Dog":@(4),
                                                    @"Bird":@(2),
                                                    }; //(2)
```

В данном примере (1) *NSArray<NSString *>* означает массив с объектами типа NSString, (2) NSDictionary<NSString *, NSNumber *> - словарь со строками в качестве ключей и объектами NSNumber в качестве значений.

Суть дженериков - проверка типа объектов на этапе компиляции. Но ведь Ojective-C язык динамический и во время выполнения что угодно можно положить куда угодно. Как с этим справляются дженерики? Если коротко, то плохо. Даже предупреждения компилятора можно получить не всегда.

OBJECTIVE-C. ДЖЕНЕРИКИ

```
NSArray<NSString *> *fakeAnimals = @[@"Cat", @(20), @«Dog"];
```

Данный код будет скомпилирован без ошибок. Встает вопрос - когда такую конструкцию можно использовать? Аккуратно и для повышения читаемости кода, главное помнить, что использование дженериков не может избавить от проверок времени выполнения.

Массивы

NSArray - это основной класс для представления массива в Objective-C, упорядоченный. Существует также его изменяемый вариант *NSMutableArray.*

Литеральный синтаксис

В Modern Syntax в Objective-C появился литеральный синтаксис, теперь массивы и словари можно объявлять с его помощью:

```
NSArray *animals = @[@"Cat",@"Dog", @«Bird"];
```

Ранее для создания массивов и словарей использовались специальные методы, например метод создания массива:

```
NSArray *animals = [[NSArray alloc] initWithObjects:@"Cat",@"Dog",
@"Bird", nil];
```

Последним в перечислении объектов идет nil.

Оценки производительности операций

- Вставка в конец массива - O(1)
- Вставка в произвольное место - O(N)
- Удаление элемента из произвольного места - O(N)
- Доступ по индексу - O(1)
- Бинарный поиск (на отсортированном массиве) - O(logN)
- Сортировка - O(n*logN)

Создание

Массивы можно создавать как с помощью литерального синтаксиса, так и с помощью конструкторов arrayWithObjects: initWithObjects: .

```
NSArray *germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche",
                         @"Opel", @"Volkswagen", @"Audi"];
NSArray *ukMakes = [NSArray arrayWithObjects:@"Aston Martin",
                    @"Lotus", @"Jaguar", @"Bentley", nil];
```

Обратиться к элементу по индексу можно с помощью синтаксиса **array**[*index*], либо с помощью метода objectAtIndex: . Первый вариант предпочтительней.

```
NSLog(@"First german make: %@", germanMakes[0]);
```

```
NSLog(@"First U.K. make: %@", [ukMakes objectAtIndex:0]);
```

Обход

Есть несколько способов обхода массивов: традиционный цикл с индексом (как в C), fast-enumeration и обход с помощью блока. Все три способа представлены ниже.

```
NSArray *germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche",
                         @"Opel", @"Volkswagen", @"Audi"];
// With fast-enumeration
for (NSString *item in germanMakes) {
    NSLog(@"%@", item);
}
// With a traditional for loop
for (int i=0; i<[germanMakes count]; i++) {
    NSLog(@"%d: %@", i, germanMakes[i]);
}
```

Обход с помощью блока.

```
[germanMakes enumerateObjectsUsingBlock:^(id obj,
                                          NSUInteger idx,
                                          BOOL *stop) {
    NSLog(@"%ld: %@", idx, obj);
}];
```

Сравнение

Массивы в Objective-C можно сравнивать с помощью метода *isEqualToArray:* , который вернет YES если размерности массивов равны и для каждой пары объектов isEqual возвращает YES.

```
NSArray *germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche",
                         @"Opel", @"Volkswagen", @"Audi"];
NSArray *sameGermanMakes = [NSArray arrayWithObjects:@"Mercedes-Benz",
                            @"BMW", @"Porsche", @"Opel",
                            @"Volkswagen", @"Audi", nil];

if ([germanMakes isEqualToArray:sameGermanMakes]) {
    NSLog(@"Oh good, literal arrays are the same as NSArrays");
}
```

Проверка элемента на вхождение

Проверить наличие объекта в массиве можно методом *containsObject:* (внутри он базируется на *indexOfObject:*). Поскольку в массиве один и тот же объект может встречаться более одного раза, а *indexOfObject:* вернет лишь первое вхождение, для решения подобных задач необходимо использовать *indexOfObject:InRange:* .

```
NSArray *germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche",
                         @"Opel", @"Volkswagen", @"Audi"];
// BOOL checking
if ([germanMakes containsObject:@"BMW"]) {
    NSLog(@"BMW is a German auto maker");
}
// Index checking
NSUInteger index = [germanMakes indexOfObject:@"BMW"];
if (index == NSNotFound) {
    NSLog(@"Well that's not quite right...");
} else {
    NSLog(@"BMW is a German auto maker and is at index %ld", index);
}
```

Не стоит забывать, что множества NSSet намного эффективней на задаче «проверка на вхождение» и по возможности нужно пользоваться ими.

Сортировка

Один из наиболее гибких способов сортировки - это сортировка с использованием метода sortedArrayUsingComparator: . Данный способ предполагает наличие блока ^NSComparisonResult(id obj1, id obj2) в котором описаны случаи:

NSOrderedAscending -> obj1 идет до obj2

NSOrderedSame для -> obj1 и obj2 порядок не важен

NSOrderedDescending -> obj1 идет после obj2

ПРИМЕР

```
NSArray *germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche",
                         @"Opel", @"Volkswagen", @"Audi"];
NSArray *sortedMakes = [germanMakes sortedArrayUsingComparator:
    ^NSComparisonResult(id obj1, id obj2) {
        if ([obj1 length] < [obj2 length]) {
            return NSOrderedAscending;
        } else if ([obj1 length] > [obj2 length]) {
            return NSOrderedDescending;
        } else {
            return NSOrderedSame;
        }
}];
NSLog(@"%@", sortedMakes);
```

Фильтрация

Фильтровать массивы возможно с использованием

filteredArrayUsingPredicate: .

```
NSArray *germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche",
                         @"Opel", @"Volkswagen", @"Audi"];

NSPredicate *beforeL = [NSPredicate predicateWithBlock:
    ^BOOL(id evaluatedObject, NSDictionary *bindings) {
        NSComparisonResult result = [@"L" compare:evaluatedObject];
        if (result == NSOrderedDescending) {
            return YES;
        } else {
            return NO;
        }
    }];
NSArray *makesBeforeL = [germanMakes
                         filteredArrayUsingPredicate:beforeL];
NSLog(@"%@", makesBeforeL);    // BMW, Audi
```

Разделение на части

```
NSArray *germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche",
```

```
                        @"Opel", @"Volkswagen", @"Audi"];

NSArray *lastTwo = [germanMakes subarrayWithRange:NSMakeRange(4, 2)];
NSLog(@"%@", lastTwo);    // Volkswagen, Audi
```

Объединение

```
NSArray *germanMakes = @[@"Mercedes-Benz", @"BMW", @"Porsche",
                         @"Opel", @"Volkswagen", @"Audi"];
NSArray *ukMakes = @[@"Aston Martin", @"Lotus", @"Jaguar", @"Bentley"];

NSArray *allMakes = [germanMakes arrayByAddingObjectsFromArray:ukMakes];
NSLog(@"%@", allMakes);
```

Reverse enumerator

Чтобы обойти массив в обратном порядке целесообразно воспользоваться методом reverseObjectEnumerator, который вернет новый развернутый массив.

ТОГДА ОБХОД БУДЕТ ВЫГЛЯДЕТЬ:

```
for (id object in [array reverseObjectEnumerator]) {
     //do something with object
}
```

Бинарный поиск

Возможно найти элемент быстрее чем за O(n) в отсортированном массиве, если использовать бинарный поиск. Для этого можно использовать метод

```
- (NSUInteger)indexOfObject:(ObjectType)obj inSortedRange:(NSRange)range
  options:(NSBinarySearchingOptions)options usingComparator:
  (NSComparator)comparator
```

С параметром NSBinarySearchingFirstEqual. Вызывать данный метод можно не только на полностью отсортированном массиве, но и на частично, для этого в range нужно указать отсортированную часть массива.

Изменяемые массивы

В изменяемые массивы можно динамически добавлять и удалять объекты.

Создание

Поскольку литеральный синтаксис позволяет создавать только неизменяемые объекты, то можно воспользоваться конструкторам arrayWithObjects: . Также существует вариант вызвать mutableCopy для созданного через литеральный конструктор неизменяемого массива.

```
NSMutableArray *brokenCars = [NSMutableArray arrayWithObjects:
                              @"Audi A6", @"BMW Z3",
                              @"Audi Quattro", @"Audi TT", nil];
```

Добавление и удаление объектов

Элементы массива можно добавить в конец, удалить последний, вставить в произвольное место, у удалить из произвольного места, заменить объект по индексу.

```
NSMutableArray *brokenCars = [NSMutableArray arrayWithObjects:
                              @"Audi A6", @"BMW Z3",
                              @"Audi Quattro", @"Audi TT", nil];
[brokenCars addObject:@"BMW F25"];
NSLog(@"%@", brokenCars);       // BMW F25 added to end
[brokenCars removeLastObject];
NSLog(@"%@", brokenCars);       // BMW F25 removed from end

// Add BMW F25 to front
[brokenCars insertObject:@"BMW F25" atIndex:0];
// Remove BMW F25 from front
[brokenCars removeObjectAtIndex:0];
// Remove Audi Quattro
[brokenCars removeObject:@"Audi Quattro»];

// Change second item to Audi Q5
[brokenCars replaceObjectAtIndex:1 withObject:@"Audi Q5"];
```

Сортировка с дескрипторами

Сортировать можно и с использованием sortUsingComparator: который работает также как и для неизменяемых массивов, здесь рассмотрим

альтернативный способ. Синтаксис данного способа несколько более компактный и удобный.

```
NSDictionary *car1 = @{
    @"make": @"Volkswagen",
    @"model": @"Golf",
    @"price": [NSDecimalNumber decimalNumberWithString:@"18750.00"]
};
NSDictionary *car2 = @{
    @"make": @"Volkswagen",
    @"model": @"Eos",
    @"price": [NSDecimalNumber decimalNumberWithString:@"35820.00"]
};
NSDictionary *car3 = @{
    @"make": @"Volkswagen",
    @"model": @"Jetta A5",
    @"price": [NSDecimalNumber decimalNumberWithString:@"16675.00"]
};
NSDictionary *car4 = @{
    @"make": @"Volkswagen",
    @"model": @"Jetta A4",
    @"price": [NSDecimalNumber decimalNumberWithString:@"16675.00"]
};
NSMutableArray *cars = [NSMutableArray arrayWithObjects:
                        car1, car2, car3, car4, nil];

NSSortDescriptor *priceDescriptor = [NSSortDescriptor
                                     sortDescriptorWithKey:@"price"
                                                 ascending:YES

selector:@selector(compare:)];
NSSortDescriptor *modelDescriptor = [NSSortDescriptor
                                     sortDescriptorWithKey:@"model"
                                     ascending:YES

selector:@selector(caseInsensitiveCompare:)];

NSArray *descriptors = @[priceDescriptor, modelDescriptor];
```

```
[cars sortUsingDescriptors:descriptors];
NSLog(@"%@", cars);    // car4, car3, car1, car2
```

Соглашение по обходу массивов

Во время обхода изменяемых массивов (любым способом) нельзя добавлять или удалять из него объекты.

Словари

NSDictionary - класс для представления словаря в Objective-C, неупорядоченный. Существует также его изменяемый вариант *NSMutableDictionary.*

Ключи словаря должны удовлетворять протоколу NSCopying (про протоколы мы вам расскажем позже).

Оценки производительности операций

• Вставка, в лучшем случае - O(1), в худшем O(N)

• Поиск по ключу, в лучшем случае - O(1)

• По значению - O(N)

• Удаление элемента - O(1)

Создание

Словари можно создать как с использование литерального синтаксиса, так используя конструкторы.

ПРИМЕР

```
// Literal syntax
NSDictionary *inventory = @{
    @"Mercedes-Benz SLK250" : [NSNumber numberWithInt:13],
    @"Mercedes-Benz E350" : [NSNumber numberWithInt:22],
    @"BMW M3 Coupe" : [NSNumber numberWithInt:19],
    @"BMW X6" : [NSNumber numberWithInt:16],
};
// Values and keys as arguments
inventory = [NSDictionary dictionaryWithObjectsAndKeys:
             [NSNumber numberWithInt:13], @"Mercedes-Benz SLK250",
             [NSNumber numberWithInt:22], @"Mercedes-Benz E350",
             [NSNumber numberWithInt:19], @"BMW M3 Coupe",
```

```
            [NSNumber numberWithInt:16], @"BMW X6", nil];
// Values and keys as arrays
NSArray *models = @[@"Mercedes-Benz SLK250", @"Mercedes-Benz E350",
                    @"BMW M3 Coupe", @"BMW X6"];
NSArray *stock = @[[NSNumber numberWithInt:13],
                   [NSNumber numberWithInt:22],
                   [NSNumber numberWithInt:19],
                   [NSNumber numberWithInt:16]];
inventory = [NSDictionary dictionaryWithObjects:stock forKeys:models];
NSLog(@"%@", inventory);
```

Доступ по ключу

Доступ по ключу осуществляется с использованием синтаксиса dictionary[key].

```
NSDictionary *inventory = @{
    @"Mercedes-Benz SLK250" : [NSNumber numberWithInt:13],
    @"Mercedes-Benz E350" : [NSNumber numberWithInt:22],
    @"BMW M3 Coupe" : [NSNumber numberWithInt:19],
    @"BMW X6" : [NSNumber numberWithInt:16],
};
NSLog(@"There are %@ X6's in stock", inventory[@"BMW X6"]);
NSLog(@"There are %@ E350's in stock",
      [inventory objectForKey:@"Mercedes-Benz E350»]);

NSDictionary *inventory = @{
    @"Mercedes-Benz SLK250" : [NSNumber numberWithInt:0],
    @"Mercedes-Benz E350" : [NSNumber numberWithInt:0],
    @"BMW M3 Coupe" : [NSNumber numberWithInt:19],
    @"BMW X6" : [NSNumber numberWithInt:16],
};
```

Обход словаря

Для обхода словарей также можно использовать fast enameration.

```
NSDictionary *inventory = @{
    @"Mercedes-Benz SLK250" : [NSNumber numberWithInt:13],
    @"Mercedes-Benz E350" : [NSNumber numberWithInt:22],
    @"BMW M3 Coupe" : [NSNumber numberWithInt:19],
    @"BMW X6" : [NSNumber numberWithInt:16],
```

```
};
NSLog(@"We currently have %ld models available", [inventory count]);
for (id key in inventory) {
    NSLog(@"There are %@ %@'s in stock", inventory[key], key);
}
```

Также можно вытащить и преобразовать в массив все ключи и все значения.

```
NSLog(@"Models: %@", [inventory allKeys]);
NSLog(@"Stock: %@", [inventory allValues]);
```

Также доступен метод обхода с помощью блоков.

```
[inventory enumerateKeysAndObjectsUsingBlock:^(id key, id obj, BOOL
*stop) {
    NSLog(@"There are %@ %@'s in stock", obj, key);
}];
```

Сравнение словарей

Работает также как и для словарей isEqualToDictionary: возвращает YES когда оба словаря имеют одинаковые пары ключ-значение.

```
NSDictionary *warehouseInventory = @{
    @"Mercedes-Benz SLK250" : [NSNumber numberWithInt:13],
    @"Mercedes-Benz E350" : [NSNumber numberWithInt:22],
    @"BMW M3 Coupe" : [NSNumber numberWithInt:19],
    @"BMW X6" : [NSNumber numberWithInt:16],
};
NSDictionary *showroomInventory = @{
    @"Mercedes-Benz SLK250" : [NSNumber numberWithInt:13],
    @"Mercedes-Benz E350" : [NSNumber numberWithInt:22],
    @"BMW M3 Coupe" : [NSNumber numberWithInt:19],
    @"BMW X6" : [NSNumber numberWithInt:16],
};
if ([warehouseInventory isEqualToDictionary:showroomInventory]) {
    NSLog(@"Why are we storing so many cars in our showroom?");
}
```

Сортировка ключей

Словарь нельзя непосредственно отсортировать в новый словарь, однако есть возможность отсортировать ключи словаря в массив использую keysSortedByValueUsingComparator: .

```
NSDictionary *prices = @{
    @"Mercedes-Benz SLK250" : [NSDecimalNumber
decimalNumberWithString:@"42900.00"],
    @"Mercedes-Benz E350" : [NSDecimalNumber
decimalNumberWithString:@"51000.00"],
    @"BMW M3 Coupe" : [NSDecimalNumber
decimalNumberWithString:@"62000.00"],
    @"BMW X6" : [NSDecimalNumber decimalNumberWithString:@"55015.00"]
};
NSArray *sortedKeys = [prices keysSortedByValueUsingComparator:
                       ^NSComparisonResult(id obj1, id obj2) {
                           return [obj2 compare:obj1];
                       }];
NSLog(@"%@", sortedKeys);
```

Фильтрация ключей

Аналогично работает фильтрация только в данном случае ключи прошедшие тест в методе keysOfEntriesPassingTest: будут записаны в множество.

```
NSDictionary *prices = @{
    @"Mercedes-Benz SLK250" : [NSDecimalNumber
decimalNumberWithString:@"42900.00"],
    @"Mercedes-Benz E350" : [NSDecimalNumber
decimalNumberWithString:@"51000.00"],
    @"BMW M3 Coupe" : [NSDecimalNumber
decimalNumberWithString:@"62000.00"],
    @"BMW X6" : [NSDecimalNumber decimalNumberWithString:@"55015.00"]
};
NSDecimalNumber *maximumPrice = [NSDecimalNumber
decimalNumberWithString:@"50000.00"];
NSSet *under50k = [prices keysOfEntriesPassingTest:
                   ^BOOL(id key, id obj, BOOL *stop) {
```

```
                    if ([obj compare:maximumPrice] ==
NSOrderedAscending) {
                        return YES;
                    } else {
                        return NO;
                    }
                }];
NSLog(@"%@", under50k);
```

Изменяемые словари

В изменяемые словари можно добавлять пары ключ-значения и можно их удалять, производительность аналогична неизменяемому словарю.

Создание

Изменяемые словари можно создавать используя метод dictionaryWithDictionary:, либо даже можно использовать mutableCopy.

```
NSMutableDictionary *jobs = [NSMutableDictionary
                             dictionaryWithDictionary:@{
                                 @"Audi TT" : @"John",
                                 @"Audi Quattro (Black)" : @"Mary",
                                 @"Audi Quattro (Silver)" : @"Bill",
                                 @"Audi A7" : @"Bill"
                             }];
NSLog(@"%@", jobs);
Добавление и удаление объектов
NSMutableDictionary *jobs = [NSMutableDictionary
                             dictionaryWithDictionary:@{
                                 @"Audi TT" : @"John",
                                 @"Audi Quattro (Black)" : @"Mary",
                                 @"Audi Quattro (Silver)" : @"Bill",
                                 @"Audi A7" : @"Bill"
                             }];
// Transfer an existing job to Mary
[jobs setObject:@"Mary" forKey:@"Audi TT"];
// Finish a job
[jobs removeObjectForKey:@"Audi A7"];
// Add a new job
jobs[@"Audi R8 GT"] = @"Jack";
```

Объединение словарей

```
NSMutableDictionary *jobs = [NSMutableDictionary
                             dictionaryWithDictionary:@{
                                 @"Audi TT" : @"John",
                                 @"Audi Quattro (Black)" : @"Mary",
                                 @"Audi Quattro (Silver)" : @"Bill",
                                 @"Audi A7" : @"Bill"
                             }];
NSDictionary *bakerStreetJobs = @{
    @"BMW 640i" : @"Dick",
    @"BMW X5" : @"Brad"
};
[jobs addEntriesFromDictionary:bakerStreetJobs];

// Create an empty mutable dictionary
NSMutableDictionary *jobs = [NSMutableDictionary dictionary];
// Populate it with initial key-value pairs
[jobs addEntriesFromDictionary:@{
    @"Audi TT" : @"John",
    @"Audi Quattro (Black)" : @"Mary",
    @"Audi Quattro (Silver)" : @"Bill",
    @"Audi A7" : @"Bill"
}];
```

Соглашение по обходу словарей

Справедливо то же правило, что и для массивов. При обходе изменяемого словаря нельзя добавлять и удалять элементы их него.

Множества

NSSet - основной класс для представления множества, коллекция неупорядоченная. Существует изменяемый вариант NSMutableSet.

Оценки производительности операций

- Вставка, в лучшем случае - O(1), в худшем O(N)
- Поиск, в лучшем случае - O(1)
- Удаление элемента - O(1)

Создание

Множество можно создать с помощью метода setWithObjects: , в конце списка объектов должен обязательно идти nil.

```
NSSet *americanMakes = [NSSet setWithObjects:@"Chrysler", @"Ford",
                                             @"General Motors", nil];
NSLog(@"%@", americanMakes);
```

Так же во многих применениях удобно создавать множество из массива.

```
NSArray *japaneseMakes = @[@"Honda", @"Mazda",
                           @"Mitsubishi", @"Honda"];
NSSet *uniqueMakes = [NSSet setWithArray:japaneseMakes];
NSLog(@"%@", uniqueMakes);    // Honda, Mazda, Mitsubishi
```

Обход

Fast enumeration, предпочтительный вариант.

```
NSSet *models = [NSSet setWithObjects:@"Civic", @"Accord",
                                      @"Odyssey", @"Pilot", @"Fit",
nil];
NSLog(@"The set has %li elements", [models count]);
for (id item in models) {
    NSLog(@"%@", item);
}
```

Обход с помощью блока:

```
[models enumerateObjectsUsingBlock:^(id obj, BOOL *stop) {
    NSLog(@"Current item: %@", obj);
    if ([obj isEqualToString:@"Fit"]) {
        NSLog(@"I was looking for a Honda Fit, and I found it!");
        *stop = YES;    // Stop enumerating items
    }
}];
```

Сравнение

Операции сравнения на NSSet аналогичны по семантике операциям сравнения на математических множествах (полное совпадение, является ли множество подмножеством, пересекаются ли множества).

```
NSSet *japaneseMakes = [NSSet setWithObjects:@"Honda", @"Nissan",
```

```
                                @"Mitsubishi", @"Toyota", nil];
NSSet *johnsFavoriteMakes = [NSSet setWithObjects:@"Honda", nil];
NSSet *marysFavoriteMakes = [NSSet setWithObjects:@"Toyota",
                                                  @"Alfa Romeo", nil];

if ([johnsFavoriteMakes isEqualToSet:japaneseMakes]) {
    NSLog(@"John likes all the Japanese auto makers and no others");
}
if ([johnsFavoriteMakes intersectsSet:japaneseMakes]) {
    // You'll see this message
    NSLog(@"John likes at least one Japanese auto maker");
}
if ([johnsFavoriteMakes isSubsetOfSet:japaneseMakes]) {
    // And this one, too
    NSLog(@"All of the auto makers that John likes are Japanese");
}
if ([marysFavoriteMakes isSubsetOfSet:japaneseMakes]) {
    NSLog(@"All of the auto makers that Mary likes are Japanese");
}
```

Проверка элемента на вхождение

```
NSSet *selectedMakes = [NSSet setWithObjects:@"Maserati",
                                             @"Porsche", nil];
// BOOL checking
if ([selectedMakes containsObject:@"Maserati"]) {
    NSLog(@"The user seems to like expensive cars");
}
// nil checking
NSString *result = [selectedMakes member:@"Maserati"];
if (result != nil) {
    NSLog(@"%@ is one of the selected makes", result);
}
```

Фильтрация

Фильтровать множество можно с использованием objectsPassingTest: и созданием нового множества из объектов прошедших тест.

```
NSSet *toyotaModels = [NSSet setWithObjects:@"Corolla", @"Sienna",
```

```
                        @"Camry", @"Prius",
                        @"Highlander", @"Sequoia", nil];
NSSet *cModels = [toyotaModels objectsPassingTest:^BOOL(id obj, BOOL
*stop) {
    if ([obj hasPrefix:@"C"]) {
        return YES;
    } else {
        return NO;
    }
}];
NSLog(@"%@", cModels);    // Corolla, Camry
```

Объединение

Множества можно объединять с использованием setByAddingObjectsFromSet: .

```
NSSet *affordableMakes = [NSSet setWithObjects:@"Ford", @"Honda",
                                    @"Nissan", @"Toyota", nil];
NSSet *fancyMakes = [NSSet setWithObjects:@"Ferrari", @"Maserati",
                                        @"Porsche", nil];
NSSet *allMakes = [affordableMakes
setByAddingObjectsFromSet:fancyMakes];
NSLog(@"%@", allMakes);
```

Изменяемые множества

Создание

Изменяемую коллекцию можно создавать явно указав объекты в конструкторе, так и создать пустое множество выбрав его исходный размер воспользовавшись методом *setWithCapacity:* . Задание capacity = 5 не означает, что в множество нельзя добавить более 5 элементов, указание емкости необходимо лишь для внутренней оптимизации структуры.

```
NSMutableSet *brokenCars = [NSMutableSet setWithObjects:
                            @"Honda Civic", @"Nissan Versa", nil];
NSMutableSet *repairedCars = [NSMutableSet setWithCapacity:5];
```

Добавление и удаление объектов

В изменяемой версии множества существуют методы для удаления и добавления объектов *addObject:* и *removeObject:* , также существует возможность удалить все объекты используя метод *removeAllObjects:* .

```
NSMutableSet *brokenCars = [NSMutableSet setWithObjects:
                            @"Honda Civic", @"Nissan Versa", nil];
NSMutableSet *repairedCars = [NSMutableSet setWithCapacity:5];
// "Fix" the Honda Civic
[brokenCars removeObject:@"Honda Civic"];
[repairedCars addObject:@"Honda Civic"];

NSLog(@"Broken cars: %@", brokenCars);     // Nissan Versa
NSLog(@"Repaired cars: %@", repairedCars); // Honda Civic
```

Фильтрация с предикатом

Не существует мутабельной версии метода *objectsPassingTest:* , однако того же самого результата можно добиться с использованием метода *filteringUsingPredicate:* . Предикаты очень мощный инструмент, рассмотрение которого выходит за граница темы коллекций (информацию можно найти в гайдах Apple).

```
NSMutableSet *toyotaModels = [NSMutableSet setWithObjects:@"Corolla",
@"Sienna",
                              @"Camry", @"Prius",
                              @"Highlander", @"Sequoia", nil];
NSPredicate *startsWithC = [NSPredicate predicateWithBlock:
                            ^BOOL(id evaluatedObject, NSDictionary
*bindings) {
                                if ([evaluatedObject hasPrefix:@"C"]) {
                                    return YES;
                                } else {
                                    return NO;
                                }
                            }];
[toyotaModels filterUsingPredicate:startsWithC];
```

```
NSLog(@"%@", toyotaModels);    // Corolla, Camry
```

Операции из теории множеств

Для изменяемых множеств существуют операции из теории множеств: объединение, пересечение, разность.

```
NSSet *japaneseMakes = [NSSet setWithObjects:@"Honda", @"Nissan",
                        @"Mitsubishi", @"Toyota", nil];
NSSet *johnsFavoriteMakes = [NSSet setWithObjects:@"Honda", nil];
NSSet *marysFavoriteMakes = [NSSet setWithObjects:@"Toyota",
                             @"Alfa Romeo", nil];

NSMutableSet *result = [NSMutableSet setWithCapacity:5];
// Объединение
[result setSet:johnsFavoriteMakes];
[result unionSet:marysFavoriteMakes];
NSLog(@"Either John's or Mary's favorites: %@", result);
// Пересечение
[result setSet:johnsFavoriteMakes];
[result intersectSet:japaneseMakes];
NSLog(@"John's favorite Japanese makes: %@", result);
//Разность
[result setSet:japaneseMakes];
[result minusSet:johnsFavoriteMakes];
NSLog(@"Japanese makes that are not John's favorites: %@",
      result);
```

Соглашение по обходу множества

Справедливо то же соглашение, что для словарей и массивов - при обходе изменяемого множества нельзя добавлять и удалять элементы из него.

Домашнее задание

Рассмотреть коллекции NSPointerArray, NSHashTable, NSMapTable, описать несколько примеров их использования, написать код, разобраться чем данные коллекции отличаются от NSArray, NSSet, NSDictionary.

Рассмотреть дополнительные виды множеств NSCountedSet, NSOrderedSet.