Урок 3

Циклы, массивы, структуры и перечисления

Массивы, указатели, циклы, структуры и перечисления. Разновидности коллекций и их отличия.

Коллекции

Коллекции

Коллекции в Objective-C применяются для хранения групп элементов (переменных) и работы с ними. Коллекции могут быть изменяемыми (Mutable) и неизменяемыми. Они содержат только объекты: переменные типов **int**, **float** и другие необходимо преобразовывать в объекты, чтобы добавить в коллекцию.

Коллекции выполняют следующие задачи:

- Перечисление объектов в коллекции;
- Определение, существует ли объект в коллекции;
- Предоставление доступа к отдельным элементам коллекции.

Изменяемые коллекции позволяют:

- Добавлять объекты в коллекцию;
- Удалять объекты из коллекции.

Каждый из видов коллекций обладает особыми характеристиками и применяется в зависимости от задачи, которая стоит перед разработчиком.

Виды коллекций

1. **Массив (NSArray и NSMutableArray) –** это упорядоченная коллекция, которая индексирует содержимое. Массив удобен для хранения значений и быстрого доступа к ним по индексу;
2. **Словарь (NSDictionary и NSMutableDictionary) –** это неупорядоченная коллекция, которая предоставляет доступ к значению посредством ключа-значения. Словари необходимы для хранения значений, которые можно сгруппировать или иным способом улучшить с помощью ключа;
3. **Множество (NSSet и NSMutableSet) –** это неупорядоченная коллекция, которая позволяет быстро добавить, удалить элемент или проверить его существование в коллекции. Множества не могут хранить несколько одинаковых объектов.

NSArray и NSMutableArray

Массивы представлены в Objective-C типами **NSArray** (неизменяемый) и **NSMutableArray** (изменяемый). Массив может содержать объекты разных типов, но не может хранить значение **nil**.

NSArray

Объект типа **NSArray** является неизменяемым массивом. Это означает, что после инициализации в него нельзя добавлять новые объекты и удалять старые. Но объекты в массиве можно изменять.

Чтобы создать массив типа **NSArray** воспользуемся одним из существующих инициализаторов. Так как массив может хранить только объекты, то перед цифрами необходимо добавить символ «**@**» – для преобразования в объекты **NSNumber**:

```
NSArray *array = [NSArray arrayWithObjects:@1, @2, @3, @4, @5, nil];
```

Так как объекты в массиве проиндексированы, к ним можно обращаться через индексы. Индексация массива выполняется с 0 и до последнего элемента. Чтобы получить доступ к элементам массива, воспользуемся специальным методом:

```
NSNumber *number = [array objectAtIndex:1]; // 2
```

Создавать массивы и получать их элементы приходится часто, поэтому есть и альтернативный путь:

```
NSArray *alterArray = @[@1, @2, @3, @4, @5]; // Создание массива
NSNumber *alterNumber = alterArray[1]; // Получение второго элемента массива
```

Для получения количества элементов в массиве можно воспользоваться методом **count**:

```
NSInteger countArray = [array count]; // 5
```

Для доступа к первому и последнему объекту существуют такие методы:

```
NSNumber *first = [array firstObject];
NSNumber *last = [array lastObject];
```

NSArray содержит еще множество методов. Мы рассмотрели основные – наиболее распространенные в работе.

NSMutableArray

В **NSMutableArray** – изменяемый массив – можно добавлять объекты и удалять их. Он автоматически выделяет или освобождает память при необходимости.

В **NSMutableArray** используются все методы, которые есть у NSArray, и дополнительно – методы для изменения элементов массива. Инициализация массива изменяемого типа также аналогична неизменяемому:

```
NSMutableArray *mutableArray = [NSMutableArray arrayWithObjects:@1, @2, @3, @4,
@5, nil];
```

Основные методы, которые отвечают за изменения массива:

- **addObject:** – добавление объекта;
- **insertObject:atIndex:** – вставка объекта по определенному индексу;
- **removeLastObject** – удаление последнего элемента массива;
- **removeObjectAtIndex:** – удаление элемента массива по индексу;
- **replaceObjectAtIndex:withObject:** – замена элемента по индексу на иной объект.

Применение основных методов NSMutableArray:

```
NSMutableArray *mutableArray = [NSMutableArray arrayWithObjects:@1, @2, @3, @4,
@5, nil];
[mutableArray addObject:@6];
[mutableArray removeObjectAtIndex:0];
[mutableArray insertObject:@7 atIndex:0];

NSLog(@"Elements: \n 0 - %@, \n 1 - %@, \n 2 - %@, \n 3 - %@, \n 4 - %@",
mutableArray[0], mutableArray[1], mutableArray[2], mutableArray[3],
mutableArray[4]);
```

Результатом выполнения программы будет строка в консоли:

```
Elements:
 0 - 7,
 1 - 2,
 2 - 3,
 3 - 4,
 4 - 5
```

NSDictionary и NSMutableDictionary

Словари хранят данные с помощью ключа-значения. Такая коллекция представляет собой записи, которые имеют уникальный ключ и значение, которое ему соответствует. Записи в словаре хранятся неупорядоченно. Значением, как и в массиве, может быть любой объект.

Ключом может быть объект, который подписан на протокол **NSCopying** и реализует методы **hash** и **isEqual:**. Чаще всего в роли ключа выступает тип **NSString**.

Словари в Cocoa представлены типами **NSDictionary** и **NSMutableDictionary**. Как и подобные массивы, первый является неизменяемым, а второй можно изменять.

Словарь применяет хэш-таблицу для хранения и быстрого доступа к значению. Но благодаря методам, которые предоставляют **NSDictionary** и **NSMutableDictionary**, разработчику не приходится работать с хэш-таблицей напрямую. Он оперирует только ключами.

NSDictionary

В **NSDictionary** после инициализации невозможно удалить или добавить значение, но можно изменять существующие.

Для создания такого словаря следует воспользоваться одним из существующих инициализаторов:

```
NSDictionary *dictionary = [NSDictionary dictionaryWithObjectsAndKeys:@"value",
@"key", @"value2", @"key2", nil];
```

Для доступа к значению применим ключ и воспользуемся специальным методом:

```
NSString *value = [dictionary valueForKey:@"key"]; // value
```

Как и у массивов, у словарей существует литеральный способ создания и доступа к значению:

```
NSDictionary *dictionary = @{ @"key": @"value", @"key2": @"value2" }; //
Создание словаря

NSString *value = dictionary[@"key"]; // Получение значения, которое
соответствует ключу key
```

Можно воспользоваться специальными методами, которые преобразуют все ключи и все значения в массив:

```
[dictionary allKeys]; // Все ключи массивом
[dictionary allValues]; // Все значения массивом
```

NSMutableDictionary

NSMutableDictionary – изменяемая версия словаря – обладает всеми особенностями и возможностями **NSDictionary**. Для его создания и обращения к его значениям присутствуют те же методы, что у неизменяемой версии.

```
NSMutableDictionary *dictionary = [NSMutableDictionary
dictionaryWithObjectsAndKeys:@"value", @"key", nil]; // Создание изменяемого
словаря
```

Для добавления и удаления значений применяются следующие функции:

```
[dictionary setValue:@"value2" forKey:@"key2"];     // Добавление значение
[dictionary removeObjectForKey:@"key"];             // Удаление значения
```

NSSet и NSMutableSet

Множество – это набор уникальных неупорядоченных элементов. Объекты внутри такой коллекции хранятся в случайном порядке, а все значения являются уникальными.

В Cocoa множества представлены типами **NSSet** и **NSMutableSet**: изменяемой и неизменяемой реализацией.

NSSet

NSSet – неизменяемая версия множества. Для инициализации применяется один из существующих конструкторов:

```
NSSet *set = [NSSet setWithObjects:@1, @2, @3, @4, @5, nil];
```

Основные методы для работы с множествами:

```
[set allObjects];        // Массив всех содержащихся в множестве объектов
[set anyObject];         // Возвращается некоторый объект из множества
[set count];             // Количество объектов в множестве
[set member:object];     // Возвращается объект в множестве, который
эквивалентен передаваемому объекту
[set intersectsSet:set]; // Проверяет два набора на разделение по крайней мере
одного объекта
[set isEqualToSet:set];  // Проверяет два множества на эквивалентность
[set isSubsetOfSet:set]; // Проверяет все объекты, содержащиеся в наборе, на
присутствие и в другом наборе
```

Рассмотрим пример, в котором проверим два множества на пересечение хотя бы одного объекта:

```
NSSet *set = [NSSet setWithObjects:@"John", @"Steve", nil];
NSSet *anotherSet = [NSSet setWithObjects:@"John", @"Alexander", nil];

BOOL result = [set intersectsSet:anotherSet];
NSLog(@"Result - %@", result ? @"YES" : @"NO");
```

Объявлены два множества с именами, среди которых пара одинаковых – John. Метод **intersectSet** определяет совпадение хотя бы одного элемента в множестве. В нашем случае программа выведет в консоль значение **YES**.

После пересечения проверим существование объекта в множестве:

```
NSSet *set = [NSSet setWithObjects:@"John", @"Steve", nil];

NSLog(@"Result - %@", [set member:@"Alexander"] ? @"YES" : @"NO");
```

Теперь будет выведено значение **NO**, так как в множестве **set** нет эквивалентного объекта для **@”Alexander”**.

NSMutableSet

NSMutableSet – изменяемая версия множества – инициализируется так же, как и неизменяемый вариант.

Основные методы для изменения множества:

```
[set addObject:object];          // Добавляет объект
[set addObjectsFromArray:array]; // Добавляет все объекты из массива
[set unionSet:anotherSet];       // Добавляет все объекты из другого набора,
уникальные для данного

[set intersectSet:anotherSet];   // Удаляет все объекты, которые не находятся в
другом наборе
[set removeAllObjects];          // Удаляет все объекты
[set removeObject:object];       // Удаляет конкретный объект
[set minusSet:anotherSet];       // Удаляет все объекты, которые находятся в
переданном множестве
```

Указатели

Указатели предоставляют косвенный доступ к значению определенного элемента данных. Они позволяют эффективно представлять сложные структуры данных, изменять значения, передаваемые в виде аргументов функциям и методам, а также проще и эффективнее работать с массивами.

Рассмотрим пример: объявим переменную, которая будет хранить число:

```
int number = 20;
```

Создадим еще одну переменную, которая будет предоставлять косвенный доступ к первой переменной:

```
int * pointer;
```

Благодаря знаку «*» компилятор понимает, что данная переменная является указателем на тип **int**. Она будет хранить ссылку на блок памяти.

```
pointer = &number;
```

В этом фрагменте **pointer** был присвоен указатель на переменную **number**. Знак «**&**» позволяет вернуть указатель на объект. При присваивании указателя в переменную (pointer) сохраняется не значение переменной, на которую он указывает, а именно сам указатель.

Чтобы получить значение, на которое он указывает, необходимо применить знак «*» перед указателем:

```
int result = * pointer; // 20
```

Так с помощью указателей можно сохранять ссылки на объекты и впоследствии применять их.

Циклы

Цикл – это многократно выполняющийся участок кода. Он применяется в программах часто, так как позволяет программисту не копировать фрагменты кода несколько раз подряд.

При использовании циклов код может выполняться заданное количество раз, либо пока соблюдается определенное условие. Для реализации в Objective-C присутствуют три вида циклов: **for**, **while** и **do-while**.

Цикл For

Цикл **for** предназначен для повторного выполнения блоков кода. Его структура выглядит так:

```
for (/* Создание переменной итерации */; /* Условие, при котором продолжать выполнение цикла */; /* Увеличение значения переменной итерации */) {
    // Повторяющийся блок кода
}
```

Пример цикла **for**:

```
for (int i = 0; i < 3; i++) {
    NSLog(@"%i", i);
}
```

При выполнении данного цикла блок кода повторяется для каждого элемента в последовательности. Перед каждым переходом к следующему элементу последовательности (итерацией) переменная становится равной этому элементу.

Также существует цикл **for** для обхода коллекций. Рассмотрим на примере:

```
NSArray *numbers = @[@1, @2, @3];
for (NSNumber *number in numbers) {
    NSLog(@"%@", number);
}
```

После выполнения цикла на экране консоли будет такой результат:

```
1
2
3
```

Сначала объявляется массив **numbers**, которому присваиваются значения 1, 2 и 3. После этого создается цикл, в котором **number** – переменная, хранящая значение для текущей итерации, а **numbers** – последовательность (массив чисел). При каждой новой итерации в консоль выводится текущее значение переменной (**number**).

Последовательностью в цикле может выступать любая коллекция. Для такого цикла переменная часть будет состоять из текущего ключа.

```
NSDictionary *dict = @{ @"key1" : @"value1", @"key2": @"value2"};
for (NSString *key in dict) {
    NSLog(@"Value - %@", [dict valueForKey:key]);
}
```

В результате выполнения в консоли будет отображен результат:

```
Value - value1
Value - value2
```

Цикл While

Оператор **while** применяется для создания циклов, которые будут выполняться при соблюдении заданного условия. В большинстве случаев циклы **while** и **for** взаимозаменяемы. Для наглядности создадим пример, который будет выполнять функцию, аналогичную **for**.

Использование цикла **while:**

```
int i = 1;
while (i <= 3) {
    NSLog(@"%i", i);
    i++;
}
```

Данный цикл **while**, как и ранее представленный **for**, выводит в консоль целые числа от 1 до 3. Разница между этими видами циклов заключается только в том, что **while** выполняется только при соблюдении условия, а **for** обходит полностью всю предоставленную последовательность.

Рассмотрим подробнее, как работает цикл **while**. Сначала объявляется переменная **i**, которая имеет значение 1. После этого объявляется цикл **while**, который каждый раз проверяет условие. Если оно истинно, то выполняется внутренний блок кода. Внутри самого блока выводится текущее значение переменной **i**, и затем происходит ее увеличение на единицу. Если забыть увеличить переменную, то **i** всегда будет удовлетворять условию, что приведет к бесконечному выполнению цикла.

Цикл Do-While

Оператор **do-while** похож на **while**. В нем так же блок кода выполняется при соблюдении условия. Разница в том, что в данном операторе сначала выполняется блок кода, и только потом происходит проверка условия.

Использование цикла **do-while**:

```
int i = 1;
do {
    NSLog(@"%i", i);
    i++;
} while (i <= 3);
```

Результат выполнения такой же, как и в предыдущем примере. Программа выводит в консоль целые числа от 1 до 3.

Операторы break и continue

Для управления циклами есть специальные слова: **break** и **continue**.

Break останавливает выполнение цикла и позволяет полностью выйти из него. **Continue** переводит на следующую итерацию без дальнейшего продолжения выполнения.

Использование **break** для цикла **for**:

```
NSArray *numbers = @[@1, @2, @3];
for (NSNumber *number in numbers) {
    if (number.integerValue == 2) {
        break;
    }
    NSLog(@"%@", number);
}
```

Выполнение цикла в данном случае прекратится после того, как **number** будет равен 2 (в текущем случае – после первого шага).

Использование **continue** для цикла **for**:

```
NSArray *numbers = @[@1, @2, @3];
for (NSNumber *number in numbers) {
    if (number.integerValue == 2) {
        continue;
    }
    NSLog(@"%@", number);
}
```

Выполнение цикла будет проходить в обычном режиме, но когда **number** будет равен 2, цикл сразу перейдет на следующую итерацию, без выполнения дальнейших строк кода в цикле. Результатом на консоли будет последовательность цифр от 1 до 3 без цифры 2.

Структуры и перечисления

Структуры

В языке Objective-C немного типов данных, и они организуются с помощью массивов, указателей, а также структур. Структуры позволяют объединить несколько типов данных в единый. К составным элементам структур можно однозначно обращаться по именам.

В Cocoa часто встречаются структуры, и некоторые типы, которые являются объектами, в них не так просто использовать. Чтобы корректно добавлять объектные типы в структуры, необходимо прописывать **__unsafe_unretained** перед типом. Этот атрибут позволяет сохранять адрес на объект. Подробнее о подобных атрибутах поговорим на уроке, посвященном управлению памятью.

Чтобы разобраться со структурами, рассмотрим пример:

```
struct Position {
    CGFloat top;
    CGFloat left;
    CGFloat right;
    CGFloat bottom;
};

typedef struct Position Position;
```

С помощью ключевого слова **struct** создаются структуры. В теле указываются параметры, которым можно будет установить значение, а также прочитать его. С помощью **typedef** созданная структура объявляется как тип. Если опустить эту строку, то при объявлении необходимо будет указывать полное название с ключевым словом **struct** (**struct Position**).

Применять структуры так же просто, как и объявлять:

```
Position position;
position.top = 10.0;
position.left = 20.0;
position.right = 20.0;
position.bottom = 10.0;

NSLog(@"Position: \n top %f, \n bottom %f, \n left %f, \n right %f",
position.top, position.bottom, position.left, position.right);
```

Таким образом мы объявили структуру и установили значения ее переменным, а позже вывели результаты в консоль:

```
Position:
top 10.000000,
bottom 10.000000,
left 20.000000,
right 20.000000
```

При объявлении можно также установить значения без необходимости обращаться к каждому, а можно указать их в фигурных скобках через запятую:

```
Position position = { 10.0, 20.0, 20.0, 10.0 };
```

Результат выполнения будет тем же.

Структуры удобны для работы с большим количеством переменных. Они позволяют сгруппировать их по определенной логике, создавать и применять, где это необходимо.

Перечисления

В разработке часто встречаются задачи, в которых необходимо реализовать различное поведение, стили, действия. Для хранения таких возможных положений существуют перечисления.

Рассмотрим на примере. Допустим, необходимо создать приложение, в котором элемент будет изменять цвет в зависимости от настроек. Для этого необходимо создать перечисление:

```
enum Color {
    ColorBlue,
    ColorRed,
    ColorGreen,
    ColorBlack,
    ColorWhite
};
```

Объявление перечислений напоминает объявление структур. Для удобного доступа можно использовать **typedef**:

```
typedef enum Color Color;
```

Теперь можно применить перечисление и создать проверку на цвет:

```
Color color = ColorBlue;
if (color == ColorBlue) {
   return [UIColor blueColor];
}
```

Перечисления могут хранить значения для определенного состояния. Например, если необходимо реализовать скорость движения:

```
enum Speed {
    SpeedSlow = 1,
    SpeedMiddle = 2,
    SpeedHigh = 3
};

typedef NSInteger Speed;
```

В **typedef** указан тип, который содержат состояния. Теперь можно применять состояние как целое число:

```
Speed speedType = SpeedHigh;

NSLog(@"speed - %li", (long)speedType);
```

В результате в консоли будет значение скорости:

```
speed - 3
```

Перечисления в Objective-C можно применять и как определение для набора флагов. Рассмотрим пример перечисления:

```
typedef UIViewAutoresizing {
    UIViewAutoresizingNone                                  = 0,
    UIViewAutoresizingFlexibleLeftMargin          = 1 << 0,
    UIViewAutoresizingFlexibleWidth                    = 1 << 1,
    UIViewAutoresizingFlexibleRightMargin        = 1 << 2,
    UIViewAutoresizingFlexibleTopMargin          = 1 << 3,
    UIViewAutoresizingFlexibleHeight                   = 1 << 4,
    UIViewAutoresizingFlexibleBottomMargin     = 1 << 5
};
```

Этот пример взят из UI-фреймворка iOS и применяется для определения изменяемых размеров представления. Каждый флаг может иметь только одно состояние из двух: установлен или не установлен. Это возможно благодаря тому, что каждый флаг представлен всего одним битом в значении, представляющем его комбинацию. Несколько флагов можно объединить поразрядным оператором **ИЛИ** (**UIViewAutoResizingFlexibleWidth** | **UIViewAutoresizingFlexibleHeight**).

Рассмотрим подробнее. Каждый элемент представлен лишь одним битом. Если представить двоичную таблицу, то каждый флаг будет иметь следующие значения:

Название флага	**Значение**					
UIViewAutoresizingFlexibleLeftMargin	0	0	0	0	0	1
UIViewAutoresizingFlexibleWidth	0	0	0	0	1	0
UIViewAutoresizingFlexibleRightMargin	0	0	0	1	0	0
UIViewAutoresizingFlexibleTopMargin	0	0	1	0	0	0
UIViewAutoresizingFlexibleHeight	0	1	0	0	0	0
UIViewAutoresizingFlexibleBottomMargin	1	0	0	0	0	0

Исходя из этой таблицы, значение для **UIViewAutoresizingFlexibleWidth** | **UIViewAutoresizingFlexibleHeight** будет равно 010010.

Перейдем к реализации проверки значения. Для этого понадобится поразрядный оператор **И**:

```
UIViewAutoresizing resize = UIViewAutoresizingFlexibleWidth |
UIViewAutoresizingFlexibleHeight;
if (resize & UIViewAutoresizingFlexibleWidth) {
    // Установлен флаг UIViewAutoresizingFlexibleWidth
}
```

Данный подход широко применяется в библиотеках, предоставляемых компанией Apple.

Практика

Создание программы, вычисляющей факториал числа

Факториал – это сумма всех чисел от 1 до n.

Необходимо реализовать программу для вычисления факториала с помощью цикла **for** и цикла **while**.

При использовании **for** необходимо создать переменную для подсчета суммы (**totalSum**) и переменную для конечного значения (**n**). После создадим цикл от 1 до переменной **n**. На каждой итерации к переменной **totalSum** необходимо прибавлять текущее значение в цикле. После выполнения цикла выведем результат в консоль.

Реализация программы для подсчета факториала с использованием цикла **for**:

```
int totalSum = 0;                    // Переменная для результата
int n = 11;                          // Конечное значение
for (int i = 1; i <= n; i++) {
    totalSum += i;                  // Увеличение значения результата на текущее
значение итерации
}
NSLog(@"Result = %i", totalSum);   // Вывод результата
```

Чтобы рассчитать факториал для другого конечного числа (не 11), необходимо поменять значение у переменной **n**.

При использовании цикла **while** необходимо, как и при **for**, создать переменную для подсчета результата (**totalSum**). После создать переменную для хранения текущего значения итерации (**current**) и переменную для конечного значения (**n**). Затем добавляем цикл с таким условием: пока текущее значение будет меньше конечного. В теле цикла мы увеличиваем значение результата на текущее значение и его само на 1. После выполнения цикла выводим результат.

Реализация программы для подсчета факториала с использованием цикла **while:**

```
int totalSum = 0;          // Переменная для результата
int current = 1;           // Текущее значение итерации
int n = 11;                // Конечное значение
while (current <= n) {
    totalSum += current; // Увеличение значения результата на текущее значение
    current++;           // Увеличение текущего значения
}
NSLog(@"Result = %i", totalSum); // Вывод результата
```

Чтобы рассчитать факториал для другого конечного числа (не 11), необходимо поменять значение у переменной **n**.

Практическое задание

1. Создать массив строк и вывести его в консоль используя цикл.
2. Улучшить калькулятор с помощью перечислений, чтобы все возможные методы (сложение, вычитание, деление, умножение) передавались в виде состояния;
3. Создать структуру Human. Со свойствами “Name” (NSString), “Age”(NSInterger), “Gander”(NS_Enum). Создать несколько экземпляров структуры и вывести их в консоль.

Дополнительные материалы

1. Стивен Кочан. «Программирование на Objective-C»;
2. Скотт Кнастер, Вакар Малик, Марк Далримпл.«Objective-C. Программирование для Mac OS X и iOS».

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Стивен Кочан. «Программирование на Objective-C»;
2. Мэтт Гэлловей. «Сила Objective-C 2.0»;
3. Скотт Кнастер, Вакар Малик, Марк Далримпл.«Objective-C. Программирование для Mac OS X и iOS».