Block

- Block(메서드가 1개있는 객체) are Objective-C object(객체)
- C, C++, Objective-C 에 추가된 확장 문법(c문법을 개조한 형태)이다.
- ^를 literal문법으로 첫 시작시 사용한다.
- 외부값을 capture해서 사용한다.
- 다른 프로그램 언어의 closures(스위프트), lambdas와 같은 역할을 한다.

Block 문법 - declare / 선언하다

- return type / block name / parameter type

Block 문법 - difinition / 정의하다

- literal문법 / argument type & name / body

Block 문법 - 사용

- number( ), 호출을 하게되면 아래의 코드가 실행됨

```
//블럭 함수 사용하여 2 * 2
//리턴타입을 지정해줄경우 return을 해주어야한다.
void (^number)(NSInteger,NSInteger) = ^(NSInteger num1, NSInteger num2) {
    NSInteger total = num1 * num2;
    NSLog(@"%ld", total);
};

NSInteger number1 = 2;
NSInteger number2 = 2;
number(number1, number2);
```

다중 파라미터, 리턴 타입

```
//다중파라미터, 리턴타입
NSInteger (^totalnumber)(NSInteger,NSInteger) = ^(NSInteger count1, NSInteger count2) {
    NSInteger totalCount = count1 + count2;
    return totalCount;
};

NSInteger countNumber1 = 3;
NSInteger countNumber2 = 3;
totalnumber(countNumber1,countNumber2);
```

Block Capture Value, Share Storage : _ _block

```
//block capture value, 캡쳐
//1. 30이 들어가지만 블럭변수값을 변경되도록 하였기때문에 84가 들어간다
//2. 로그를 찍는다 84로 나온다
//3. 100이 들어간다.
//4. 로그를 찍는다. 100으로 나온다.
- (void)testMethod {

    //__block = readonly, readwrite라고 생각하면됨
    //__block은 블럭변수의 값들을 바꿀수가 있다.
    __block NSInteger anInteger = 30;

    void (^testBlock)(void) = ^{
        NSLog(@"block in integer is : %ld", anInteger);
        anInteger = 100;
    };

    anInteger = 84;

    testBlock();
    NSLog(@"block out integer is : %ld", anInteger);
}
```

Block With Typedef

- typedef 지정

```
//block을 별칭 지정
typedef void (^plus)(void);
```

- typedef 지정을 프로퍼티 사용

```
//typedef를 사용한 프로퍼티
@property plus pluscount;
```

- block에 사용

```
//1 + 1, typedef를 사용함
self.pluscount = ^{
    NSInteger plusnumber = 1 + 1;
    NSLog(@"%ld", plusnumber);
};

self.pluscount();
```

Block is Parameter

```
//다중파라미터, 리턴타입
NSInteger (^totalnumber)(NSInteger,NSInteger) = ^(NSInteger count1, NSInteger count2) {
    NSInteger totalCount = count1 + count2;
    return totalCount;
};

NSInteger countNumber1 = 3;
NSInteger countNumber2 = 3;
totalnumber(countNumber1,countNumber2);
```

Animation

UIView Animation

- 특정 시간 동안 View의 속성값을 변화시키는 작업
  예) move, fade, size change, repeat등

UIView Animation Method

```
//애니메이션
[UIView beginAnimations:@"movemove" context:nil];

[UIView setAnimationDelay:1];
[UIView setAnimationDuration:3];
[UIView setAnimationRepeatCount:3];

[self.testlabel setFrame:CGRectMake(100, 300, self.testlabel.bounds.size.width,
                                               self.testlabel.bounds.size.height)];
[self.testlabel setAlpha:0.5];

[UIView commitAnimations];

//weak으로 지정해줌
ViewController __weak *wself = self;

//block사용 애니메이션
//블럭내에서는 reperence count를 올려주기만 하고 내려주진 않기때문에 위의 self를 weak으로 지정해준다.
void (^ani)(void) = ^{
  [wself.testlabel setFrame:CGRectMake(100, 300, self.testlabel.bounds.size.width,
                                                  self.testlabel.bounds.size.height)];
};

//위의 block을 ani에 해주어도 된다.
[UIView animateWithDuration:3 animations:ani];
```

Animateble UIView Properties

- frame, bounds, center, transform, alpha, background color, content stretch

Animation 속성

- duration : animation 진행 시간
- delay : 대기 시간
- options : animation 옵션
- animations : 애니메이션 동작 block 함수
- completions : 애니메이션 완료 후 동작 block 함수

Thread

- 스레드는 어떠한 프로그램 내에서, 특히 프로세스 내에서 실행되는 흐름의 단위를 말한다. 일반적으로 한 프로그램은 하나의 스레드를 가지고 있지만, 프로그램 환경에 따라 둘 이상의 스레드를 동시에 실행할 수 있다. 이러한 실행 방식을 멀티스레드라고 한다.

둘 이상의 스레드를 멀티 스레드

iOS thread

- 모든 app은 기본적으로 main thread를 가지고 있다.
- use UIKIT classes only from your app’s main thread
- 기본적인 UI 및 모든 행동은 main thread(프로세스 내에서 실행되는 작업)에서 실행된다.

Single Thread

<process>
Thread #1

행동 1	행동 2	행동 3	행동 4

시간흐름

Multi Thread

<process>
Thread #1

행동 1	행동 2	행동 3	행동 4

시간흐름

Thread #2

행동 1	행동 2	행동 3	

시간흐름

- 동시에 작업이 필요한 경우
  - multi core process를 사용하기 위해
  - 중요한 작업에 방해를 받지 않기 위해
  - 상태를 계속 감시 해야 할 경우가 필요할때
  - 다른작업을 하면서도 또 다른작업을 할수있는...
- multi thread 사용 예
  - network request / response
  - time control
  - image download / upload
  - long time actions

동기 방식 / 비동기 방식

- 비동기방식 / 2개의 스레드에서 동시에 일어날수있다. 2줄씩
- 동기방식 / 한 스레드에서 일어난다. 순차적으로 , 한줄씩
- 디자인패턴에 의한 비동기처리는 다음과 같은 것들이 있다.
  - 델리게이트, 셀렉터, 블록, 노티피케이션
- 큐를 이용한 비동기처리 방법은 GCD로 가능하다

교착상태(deadlock)

- 교착상태란 두개 이상의 작업이 서로 상대방의 작업이 끝나기 만을 기다리고 있기 때문에 결과적으로 아무것도 완료되지 못하는 상태를 가리킨다.
- atomic : 프로퍼티의 순서를 매겨 우선순위 대로 처리한다.
- nonatomic : 순서에 상관없이 완료될 경우 무조건 처리한다.

Multithread

when is use

- 동시에 작업이 필요한 경우
- multi core process를 사용하기 위해
- 중요한 작업에 방해를 받지 않기 위해
- 상태를 계속 감시 해야 할 경우가 필요할때

ios MultiThread방법

- NSThread : 직접 thread를 만들어서 제어 하는 방식
- GCD : block 기반의 기법으로 코드 가독성이 좋고 간편하다.
- NSOperation : GCD기반의 rapper class. 간단하게 사용가능하고 고수준의 API를 제공한다. 성능이 느린편
- performselector : selector를 이용한 방식, ARC이전에 주로 사용한 방식이였으나 GCD이후엔 많이 사용되진 않는다.
- NSTimer : 간단한 interval Nofification를 제공해 주는 class, 특이하게도 NSTimer는 mainloop에서 실행된다.

NSThread

- main thread 외 다른 스레드를 만드는 클래스
- UI는 절대 추가 Thread에서 실행시키면 안된다.
- selector로 실행된 method가 종료후 자연스럽게 thread도 종료 된다.
- cancel 명령으로 강제 종료가 되지는 않는다.

```
-(IBAction)timeStart:(id)sender
{
    //스레드 생성
    self.thread = [[NSThread alloc] initWithTarget:self selector:@selector(startThread) object:nil];
    //스레드 시작
    [self.thread start];

}

//스레드 시작
- (void)startThread
{
    for (NSInteger i =0; i < 100; i ++) {

        [self performSelectorOnMainThread:@selector(changelabel) withObject:nil waitUntilDone:nil];
        sleep(1);
    }
}
```

GCD(Grand Central Dispatch)

- 비동기로 여러작업을 수행시키는 강력하고 쉬운 방법이다.
- dispatch queue를 이용해 작업들을 컨트롤 한다.
- block를 활용해서 구현한다.

dispatch queue

- dispatch queue는 GCD의 핵심으로 GCD로 실행한 작업들을 관리하는 queue이다.
- 모든 dispatch queue는 first-in, first-out 데이터 구조이다.
- dispatch queue는 크게 serial dispatch queue, concurrent dispatch queue, 2종류로 나눌수 있다.

구조적 분류

- main dispatch queue (main thread)
- global queue (system base queue)
- private queue (custom queue)

Main dispatch queue

- main thread를 가르키며 기본 UI를 제어하는 queue이다.
- serial방식으로 들어온 순서대로 진행되며 앞에 작업이 종료될때까지 뒤의 작업들은 대기 한다.

Global dispatch queue

- app 전역에 사용되는 queue로서 concurrent방식의 queue이다.
- 총 4개의 queue로 이뤄져 있으며 중요도에 따라 high, default, low, background queue중 선택되어 실행된다.

```
//GCD
//high, default, low, background
dispatch_queue_t queue = dispatch_get_main_queue();
dispatch_queue_t global = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_sync(dispatch_get_main_queue(), ^{
    for (NSInteger i = 0; i < 10000; i ++) {
        for (NSInteger j = 0; j < 10000; j ++) {

        }
    NSLog(@"main queue %ld", i);
    }
});
```

Private dispatch queue

- 사용자 정의 queue이다.
- 사용자가 만든 queue는 serial방식과 concurrent방식으로 만들 수 있다.

GCD함수

```
//serial = 직렬 큐, async = 비동기방식
dispatch_queue_t queuequeue = dispatch_queue_create("test", DISPATCH_QUEUE_SERIAL);
dispatch_async(queuequeue, ^{NSLog(@"serial async call 1");});
dispatch_async(queuequeue, ^{NSLog(@"serial async call 2");});
dispatch_async(queuequeue, ^{NSLog(@"serial async call 3");});
dispatch_async(queuequeue, ^{NSLog(@"serial async call 4");});
dispatch_async(queuequeue, ^{NSLog(@"serial async call 5");});

//concurrent = 병렬 큐, async = 비동기방식
dispatch_queue_t queuequeue1 = dispatch_queue_create("test1", DISPATCH_QUEUE_CONCURRENT);
dispatch_async(queuequeue1, ^{NSLog(@"concurrent async call 1");});
dispatch_async(queuequeue1, ^{NSLog(@"concurrent async call 2");});
dispatch_async(queuequeue1, ^{NSLog(@"concurrent async call 3");});
dispatch_async(queuequeue1, ^{NSLog(@"concurrent async call 4");});
dispatch_async(queuequeue1, ^{NSLog(@"concurrent async call 5");});

//concurrent = 병렬 큐, sync = 동기방식
dispatch_queue_t queuequeue2 = dispatch_queue_create("test1", DISPATCH_QUEUE_CONCURRENT);
dispatch_sync(queuequeue2, ^{NSLog(@"concurrent sync call 1");});
dispatch_sync(queuequeue2, ^{NSLog(@"concurrent sync call 2");});
dispatch_sync(queuequeue2, ^{NSLog(@"concurrent sync call 3");});
dispatch_sync(queuequeue2, ^{NSLog(@"concurrent sync call 4");});
dispatch_sync(queuequeue2, ^{NSLog(@"concurrent sync call 5");});

//concurrent = 병렬 큐, after = 시간이 지난 후 작업을 추가한다.
dispatch_queue_t queuequeue3 = dispatch_queue_create("test3", DISPATCH_QUEUE_CONCURRENT);
dispatch_time_t popTime = dispatch_time(DISPATCH_TIME_NOW, 3 * NSEC_PER_SEC);
dispatch_after(popTime, queuequeue3, ^{NSLog(@"after call 1");});
dispatch_async(queuequeue3, ^{NSLog(@"async call 2");});
dispatch_async(queuequeue3, ^{NSLog(@"async call 3");});
dispatch_async(queuequeue3, ^{NSLog(@"async call 4");});
dispatch_async(queuequeue3, ^{NSLog(@"async call 5");});
dispatch_async(queuequeue3, ^{NSLog(@"async call 6");});
dispatch_async(queuequeue3, ^{NSLog(@"async call 7");});

//concurrent = 병렬 큐, barrier = 작업을 구분
dispatch_queue_t queuequeue4 = dispatch_queue_create("test4", DISPATCH_QUEUE_CONCURRENT);
dispatch_async(queuequeue4, ^{NSLog(@"async call 2");});
dispatch_async(queuequeue4, ^{NSLog(@"async call 3");});
dispatch_async(queuequeue4, ^{NSLog(@"async call 4");});
dispatch_async(queuequeue4, ^{NSLog(@"async call 5");});
dispatch_async(queuequeue4, ^{NSLog(@"async call 6");});
dispatch_async(queuequeue4, ^{NSLog(@"async call 7");});
dispatch_barrier_async(queuequeue4, ^{NSLog(@"---------------------------");});
dispatch_async(queuequeue4, ^{NSLog(@"async call 8");});
dispatch_async(queuequeue4, ^{NSLog(@"async call 9");});
dispatch_async(queuequeue4, ^{NSLog(@"async call 10");});
dispatch_async(queuequeue4, ^{NSLog(@"async call 11");});
dispatch_async(queuequeue4, ^{NSLog(@"async call 12");});
dispatch_async(queuequeue4, ^{NSLog(@"async call 13");});
```