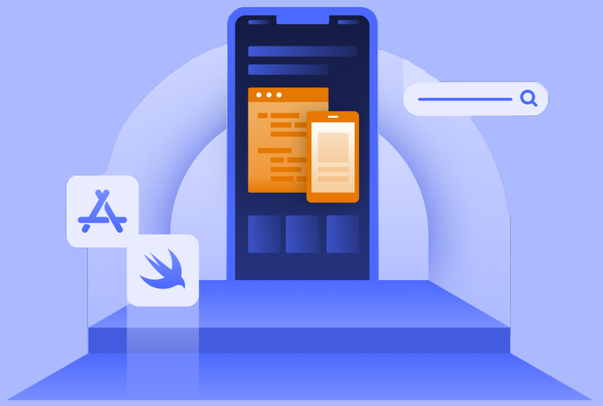
Objective-C

둘러보기



2022년 10월 11일

LIKELION APP SCHOOL

구조체

구조체

Structures

- Objective-C 배열을 사용하면 같은 종류의 여러 데이터 항목을 보유할 수 있는 변수 유형을 정의할 수 있지만 구조체는 다른 종류의 데이터 항목을 결합할 수 있는 Objective-C 프로그래밍에서 사용할 수 있는 또 다른 사용자 정의 데이터 유형입니다.

- 구조체는 레코드를 나타내는 데 사용됩니다. 도서관에서 책을 추적하고 싶다고 가정해 보겠습니다. 각 책에 대한 다음 속성을 추적할 수 있습니다.
  - 제목
  - 작가
  - 주제
  - 도서 ID

구조체 정의

Defining a Structure

- **구조체를 정의하려면 struct** 문을 사용해야 합니다.
- struct 문은 프로그램에 대해 둘 이상의 멤버가 있는 새 데이터 유형을 정의합니다. struct 문의 형식은 다음과 같습니다.

```
struct [structure tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

구조체 정의

Defining a Structure

- **구조체 태그** 는 선택 사항이며 각 멤버 정의는 int i와 같은 일반 변수 정의입니다.
- 또는 float f; 또는 다른 유효한 변수 정의. 구조 정의 끝에서 마지막 세미콜론 앞에 하나 이상의 구조 변수를 지정할 수 있지만 선택 사항입니다.
- Book 구조를 선언하는 방법은 다음과 같습니다.

```
struct Books {
   NSString *title;
   NSString *author;
   NSString *subject;
   int    book_id;
} book;
```

구조체 멤버 접근

Accessing Structure Members

- 구조체의 멤버에 액세스하려면 **멤버 액세스 연산자(.)** 를 사용 합니다.
- 멤버 액세스 연산자는 구조 변수 이름과 액세스하려는 구조 멤버 사이의 마침표로 코딩됩니다.
- 구조체 유형의 변수를 정의하려면 struct 키워드를 사용 합니다 .

```
#import <Foundation/Foundation.h>

struct Books {
   NSString *title;
   NSString *author;
   NSString *subject;
   int   book_id;
};

int main() {
   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   Book1.title = @"Objective-C Programming";
   Book1.author = @"Nuha Ali";
   Book1.subject = @"Objective-C Programming Tutorial";
   Book1.book_id = 6495407;

   /* print Book1 info */
   NSLog(@"Book 1 title : %@\n", Book1.title);
   NSLog(@"Book 1 author : %@\n", Book1.author);
   NSLog(@"Book 1 subject : %@\n", Book1.subject);
   NSLog(@"Book 1 book_id : %d\n", Book1.book_id);

   /* print Book2 info */
   NSLog(@"Book 2 title : %@\n", Book2.title);
   NSLog(@"Book 2 author : %@\n", Book2.author);
   NSLog(@"Book 2 subject : %@\n", Book2.subject);
   NSLog(@"Book 2 book_id : %d\n", Book2.book_id);

   return 0;
}
```

```
Book 1 title : Objective-C Programming
Book 1 author : Nuha Ali
Book 1 subject : Objective-C Programming Tutorial
```

함수의 인수로서의 구조체

Structures as Function Arguments

- 다른 변수나 포인터를 전달할 때와 매우 유사한 방식으로 구조체를 함수 인수로 전달할 수 있습니다.
- 앞의 예에서 액세스한 것과 유사한 방식으로 구조 변수에 액세스합니다.

```
#import <Foundation/Foundation.h>

struct Books {
   NSString *title;
   NSString *author;
   NSString *subject;
   int   book_id;
};

@interface SampleClass:NSObject
/* function declaration */
- (void) printBook:( struct Books) book ;
@end

@implementation SampleClass

- (void) printBook:( struct Books) book {
   NSLog(@"Book title : %@\n", book.title);
   NSLog(@"Book author : %@\n", book.author);
   NSLog(@"Book subject : %@\n", book.subject);
   NSLog(@"Book book_id : %d\n", book.book_id);
}

@end
```

```
int main() {
   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   Book1.title = @"Objective-C Programming";
   Book1.author = @"Nuha Ali";
   Book1.subject = @"Objective-C Programming Tutorial";
   Book1.book_id = 6495407;

   /* book 2 specification */
   Book2.title = @"Telecom Billing";
   Book2.author = @"Zara Ali";
   Book2.subject = @"Telecom Billing Tutorial";
   Book2.book_id = 6495700;

   SampleClass *sampleClass = [[SampleClass alloc]init];
   /* print Book1 info */
   [sampleClass printBook: Book1];

   /* Print Book2 info */
   [sampleClass printBook: Book2];

   return 0;
}
```

```
Book title : Objective-C Programming
Book author : Nuha Ali
Book subject : Objective-C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

구조체에 대한 포인터

Pointers to Structures

- 다음과 같이 다른 변수에 대한 포인터를 정의하는 것과 매우 유사한 방식으로 구조에 대한 포인터를 정의할 수 있습니다.

```
struct Books *struct_pointer;
```

- 이제 위에서 정의한 포인터 변수에 구조체 변수의 주소를 저장할 수 있습니다.
- 구조 변수의 주소를 찾으려면 다음과 같이 구조 이름 앞에 & 연산자를 배치합니다.

```
struct_pointer = &Book1;
```

- 해당 구조에 대한 포인터를 사용하여 구조의 멤버에 액세스하려면 다음과 같이 -> 연산자를 사용해야 합니다.

```
struct_pointer->title;
```

구조체에 대한 포인터

Pointers to Structures

```
#import <Foundation/Foundation.h>

struct Books {
   NSString *title;
   NSString *author;
   NSString *subject;
   int   book_id;
};

@interface SampleClass:NSObject
/* function declaration */
- (void) printBook:( struct Books *) book ;
@end

@implementation SampleClass
- (void) printBook:( struct Books *) book {
   NSLog(@"Book title : %@\n", book->title);
   NSLog(@"Book author : %@\n", book->author);
   NSLog(@"Book subject : %@\n", book->subject);
   NSLog(@"Book book_id : %d\n", book->book_id);
}

@end
```

```
int main() {
   struct Books Book1;        /* Declare Book1 of type Book */
   struct Books Book2;        /* Declare Book2 of type Book */

   /* book 1 specification */
   Book1.title = @"Objective-C Programming";
   Book1.author = @"Nuha Ali";
   Book1.subject = @"Objective-C Programming Tutorial";
   Book1.book_id = 6495407;

   /* book 2 specification */
   Book2.title = @"Telecom Billing";
   Book2.author = @"Zara Ali";
   Book2.subject = @"Telecom Billing Tutorial";
   Book2.book_id = 6495700;

   SampleClass *sampleClass = [[SampleClass alloc]init];
   /* print Book1 info by passing address of Book1 */
   [sampleClass printBook:&Book1];

   /* print Book2 info by passing address of Book2 */
   [sampleClass printBook:&Book2];

   return 0;
}
```

```
Book title : Objective-C Programming
Book author : Nuha Ali
Book subject : Objective-C Programming Tutorial
Book book_id : 6495407
Book title : Telecom Billing
Book author : Zara Ali
Book subject : Telecom Billing Tutorial
Book book_id : 6495700
```

비트 필드

Bit Fields

- 비트 필드를 사용하면 구조에서 데이터를 패킹할 수 있습니다. 이것은 메모리나 데이터 저장이 중요할 때 특히 유용합니다. 대표적인 예 -
  - 여러 개체를 기계어로 묶는 것. 예를 들어 1비트 플래그를 압축할 수 있습니다.
  - 외부 파일 형식 읽기 -- 비표준 파일 형식을 읽을 수 있습니다. 예: 9비트 정수.

- Objective-C를 사용하면 변수 뒤에 :bit length를 넣어 구조 정의에서 이를 수행할 수 있습니다.

비트 필드

Bit Fields

```
struct packed_struct {
   unsigned int f1:1;
   unsigned int f2:1;
   unsigned int f3:1;
   unsigned int f4:1;
   unsigned int type:4;
   unsigned int my_int:9;
} pack;
```

- 여기에서 Packed_struct는 6개의 멤버를 포함합니다
  - 4개의 1비트 플래그 f1..f3, 4비트 유형 및 9비트 my_int.
- Objective-C는 필드의 최대 길이가 컴퓨터의 정수 단어 길이보다 작거나 같은 경우 위의 비트 필드를 가능한 한 압축하여 자동으로 압축합니다.
- 그렇지 않은 경우 일부 컴파일러는 필드에 대한 메모리 겹침을 허용하는 반면 다른 컴파일러는 다음 단어에 다음 필드를 저장할 수 있습니다.

전처리기

전처리기

Preprocessors

- **Objective-C 전처리기는** 컴파일러 의 일부가 아니지만 컴파일 프로세스의 별도 단계입니다.
- 간단히 말해서 Objective-C 전처리기는 텍스트 대체 도구이며 실제 컴파일 전에 필요한 전처리를 수행하도록 컴파일러에 지시합니다.
- Objective-C 전처리기를 OCPP라고 부를 것입니다.

- 모든 전처리기 명령은 파운드 기호(#)로 시작합니다. 공백이 아닌 첫 번째 문자여야 하며 가독성을 위해 전처리기 지시문은 첫 번째 열에서 시작해야 합니다.

전처리기

Preprocessors

전처리기	설명
#define	정의된 내용으로 대체하는 전처리기 매크로입니다.
#include	다른 파일의 특정 헤더를 삽입합니다.
#undef	전처리기 매크로 정의를 해제합니다.
#ifdef	이 매크로가 정의되어 있으면 true를 반환합니다.
#ifndef	이 매크로가 정의되지 않은 경우 true를 반환합니다.
#if	컴파일 시간 조건이 참인지 테스트합니다.

전처리기

Preprocessors

전처리기	설명
#else	#if의 조건이 맞지 않을 경우 대안을 처리합니다.
#elif	#else 처리기에 #if 조건을 결합한 것입니다.
#endif	전처리기 조건부를 종료합니다.
#error	stderr에 오류 메시지를 출력합니다.
#pragma	표준화된 방법을 사용하여 컴파일러에 특수 명령을 일으킵니다.

전처리기 예제

Preprocessors Examples

- 다음 예를 통해 다양한 전처리기를 알아봅시다.

```
#define MAX_ARRAY_LENGTH 20
```

- 위 지시문은 OCPP에 MAX_ARRAY_LENGTH의 인스턴스를 20으로 바꾸도록 지시합니다.
- 가독성을 높이려면 상수에 *#define* 을 사용하세오.

전처리기 예제

Preprocessors Examples

```
#import <Foundation/Foundation.h>
#include "myheader.h"
```

- 이 지시문은 OCPP에게 **Foundation Framework** 에서 Foundation.h를 가져와 현재 소스 파일에 텍스트를 추가하도록 지시합니다.
- 다음 줄은 OCPP에게 로컬 디렉토리에서 **myheader.h** 를 가져와서 현재 소스 파일에 내용을 추가하도록 지시합니다.

전처리기 예제

Preprocessors Examples

```
#undef  FILE_SIZE
#define FILE_SIZE 42
```

- 이것은 OCPP에 기존 FILE_SIZE를 정의 해제하고 42로 정의하도록 지시합니다.

전처리기 예제

Preprocessors Examples

```
#ifndef MESSAGE
   #define MESSAGE "You wish!”
#endif
```

- 이것은 MESSAGE가 아직 정의되지 않은 경우에만 MESSAGE를 정의하도록 OCPP에 지시합니다.

전처리기 예제

Preprocessors Examples

```
#ifdef DEBUG
  /* Your debugging statements here */
#endif
```

- 이것은 DEBUG가 정의된 경우 동봉된 명령문을 처리하도록 OCPP에 지시합니다. 이것은 컴파일 시 *-DDEBUG* 플래그를 gcc 컴파일러에 전달하는 경우에 유용합니다. 이것은 DEBUG를 정의하므로 컴파일하는 동안 즉시 디버깅을 켜고 끌 수 있습니다.

미리 정의된 매크로

Predefined Macros

매크로	설명
__DATE__	"MMM DD YYYY" 형식의 문자 리터럴로서의 현재 날짜
__TIME__	"HH:MM:SS" 형식의 문자 리터럴로서의 현재 시간
__FILE__	여기에는 현재 파일 이름이 문자열 리터럴로 포함됩니다.
__LINE__	여기에는 현재 줄 번호가 10진수 상수로 포함됩니다.
__STDC__	컴파일러가 ANSI 표준을 준수하는 경우 1로 정의됩니다.

전처리기 예제

Preprocessors Examples

```
#import <Foundation/Foundation.h>

int main() {
   NSLog(@"File :%s\n", __FILE__ );
   NSLog(@"Date :%s\n", __DATE__ );
   NSLog(@"Time :%s\n", __TIME__ );
   NSLog(@"Line :%d\n", __LINE__ );
   NSLog(@"ANSI :%d\n", __STDC__ );

   return 0;
}
```

```
File :main.m
Date :Oct 7 2022
Time :04:46:14
Line :8
ANSI :1
```

전처리기 연산자

Preprocessors Operators

- Objective-C 전처리기는 매크로 생성에 도움이 되는 다음 연산자를 제공합니다.

- 매크로 연속 (\ )
  - 매크로는 일반적으로 한 줄에 포함되어야 합니다.
  - 매크로 연속 연산자는 한 줄에 너무 긴 매크로를 계속하는 데 사용됩니다.

```
#define  message_for(a, b)  \
   NSLog(@#a " and " #b ": We love you!\n")
```

전처리기 연산자

Preprocessors Operators

- 문자열화(#)
  - 문자열화 또는 숫자 기호 연산자('#')는 매크로 정의 내에서 사용될 때 매크로 매개변수를 문자열 상수로 변환합니다.
  - 이 연산자는 지정된 인수 또는 매개변수 목록이 있는 매크로에서만 사용할 수 있습니다.

```
#import <Foundation/Foundation.h>

#define  message_for(a, b)  \
   NSLog(@#a " and " #b ": We love you!\n”)

int main(void) {
   message_for(Carole, Debra);
   return 0;
}
```

```
Carole and Debra: We love you!
```

전처리기 연산자

Preprocessors Operators

- 토큰 붙여넣기(##)
  - 매크로 정의 내의 토큰 붙여넣기 연산자(##)는 두 인수를 결합합니다.
  - 매크로 정의에 있는 두 개의 개별 토큰을 단일 토큰으로 결합할 수 있습니다.

```
#import <Foundation/Foundation.h>

#define tokenpaster(n) NSLog (@"token" #n " = %d", token##n)

int main(void) {
  int token34 = 40;

  tokenpaster(34);     // NSLog (@"token34 = %d", token34);
  return 0;
}
```

```
token34 = 40
```

전처리기 연산자

Preprocessors Operators

- 정의된 ( ) 연산자
  - 전처리기 **정의** 연산자는 식별자가 #define을 사용하여 정의되었는지 확인하기 위해 상수 표현식에서 사용됩니다.
  - 지정된 식별자가 정의된 경우 값은 true(0이 아님)입니다.
  - 기호가 정의되지 않은 경우 값은 거짓(영)입니다. 정의된 연산자는 다음과 같이 지정됩니다.

```
#import <Foundation/Foundation.h>

#if !defined (MESSAGE)
   #define MESSAGE "You wish!"
#endif

int main(void) {
   NSLog(@"Here is the message: %s\n", MESSAGE);
   return 0;
}
```

```
Here is the message: You wish!
```

전처리기 연산자

Preprocessors Operators

- 매개변수화된 매크로
  - OCPP의 강력한 기능 중 하나는 매개변수화된 매크로를 사용하여 기능을 시뮬레이션하는 기능입니다.
  - 예를 들어 다음과 같이 숫자를 제곱하는 코드가 있을 수 있습니다.

```
int square(int x) {
  return x * x;
}
```

  - 다음과 같이 매크로를 사용하여 위의 코드를 다시 작성할 수 있습니다.

```
#define square(x) ((x) * (x))
```

전처리기 연산자

Preprocessors Operators

- 인수가 있는 매크로는 사용 하기 전에 **#define 지시문을 사용하여 정의해야 합니다.**
- 인수 목록은 괄호로 묶여 있으며 매크로 이름 바로 뒤에 와야 합니다.
- 매크로 이름과 여는 괄호 사이에는 공백이 허용되지 않습니다.

```
#import <Foundation/Foundation.h>

#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void) {
   NSLog(@"Max between 20 and 10 is %d\n", MAX(10, 20));
   return 0;
}
```

```
Max between 20 and 10 is 20
```

typedef

typedef

- **Objective-C 프로그래밍 언어는 typedef** 라는 키워드 를 제공하여 유형에 새 이름을 지정하는 데 사용할 수 있습니다.
- 다음은 1바이트 숫자에 대해 **BYTE** 라는 용어를 정의하는 예입니다.

```
typedef unsigned char BYTE;
```

- 이 유형 정의 후에 식별자 BYTE를 unsigned char 유형의 약어로 사용할 수 있습니다.

```
BYTE  b1, b2;
```

typedef

- 관례에 따라 대문자는 이러한 정의에 사용되어 사용자에게 유형 이름이 실제로는 기호 약어임을 상기시키지만 다음과 같이 소문자를 사용할 수 있습니다

```
typedef unsigned char byte;
```

- **typedef** 를 사용하여 사용자 정의 데이터 유형에도 이름을 지정할 수 있습니다 .
- 예를 들어, 구조체와 함께 typedef를 사용하여 새 데이터 형식을 정의한 다음 해당 데이터 형식을 사용하여 다음과 같이 직접 구조체 변수를 정의할 수 있습니다.

typedef

```objc
#import <Foundation/Foundation.h>

typedef struct Books {
   NSString *title;
   NSString *author;
   NSString *subject;
   int book_id;
} Book;

int main() {
   Book book;
   book.title = @"Objective-C Programming";
   book.author = @"TutorialsPoint";
   book.subject = @"Programming tutorial";
   book.book_id = 100;

   NSLog( @"Book title : %@\n", book.title);
   NSLog( @"Book author : %@\n", book.author);
   NSLog( @"Book subject : %@\n", book.subject);
   NSLog( @"Book Id : %d\n", book.book_id);

   return 0;
}
```

```
Book title : Objective-C Programming
Book author : TutorialsPoint
Book subject : Programming tutorial
Book Id : 100
```

typedef와 #define 비교

typedef vs #define

- **#define** 은 Objective-C 지시문으로 **typedef** 와 유사 하지만 다음과 같은 차이점이 있는 다양한 데이터 유형에 대한 별칭을 정의하는 데 사용됩니다.
  - **typedef** 는 유형에만 기호 이름을 부여하는 것으로 제한되는 반면 #define **은** 1을 ONE으로 정의할 수 있는 것처럼 값의 별칭을 정의하는 데에도 사용할 수 있습니다.
  - **typedef** 해석은 as #define 문이 전처리기에 의해 처리되는 컴파일러에 의해 수행됩니다.

typedef와 #define 비교

```
#import <Foundation/Foundation.h>

#define TURE 1
#define FALSE 0

int main() {
  NSLog( @“Value of TURE : %d\n", TRUE);
  NSLog( @“Value of FALSE : %d\n", FALSE);

   return 0;
}
```

```
Value of TRUE : 1
Value of FALSE : 0
```

타입 캐스팅

타입캐스팅

Type Casting

- 타입 캐스팅은 한 데이터 유형에서 다른 데이터 유형으로 변수를 변환하는 방법입니다.
- 예를 들어, 긴 값을 간단한 정수에 저장하려면 long to int를 입력할 수 있습니다.
- 다음과 같이 **캐스트 연산자** 를 사용하여 명시적으로 값을 한 유형에서 다른 유형으로 변환할 수 있습니다 .

```
(type_name) expression
```

- Objective-C에서는 일반적으로 32비트의 경우 float의 기본 유형에서 파생된 부동 소수점 연산을 수행하기 위해 CGFloat를 사용하고 64비트의 경우 double을 사용합니다.
- 다음은 캐스트 연산자가 하나의 정수 변수를 다른 정수 변수로 나누는 것이 부동 소수점 연산으로 수행되도록 하는 예입니다.

타입캐스팅

Type Casting

```
#import <Foundation/Foundation.h>

int main() {
   int sum = 17, count = 5;
   CGFloat mean;

   mean = (CGFloat) sum / count;
   NSLog(@"Value of mean : %f\n", mean );

   return 0;
}
```

```
Value of mean : 3.400000
```

- 여기서 캐스트 연산자가 나눗셈보다 우선하므로 **sum 값** 은 먼저 **double** 유형으로 변환 되고 마지막으로 count로 나누어 double 값을 생성한다는 점에 유의해야 합니다.
- **유형 변환은 컴파일러에 의해 자동으로 수행되는 암시적이거나 캐스트 연산자** 를 사용하여 명시적으로 지정될 수 있습니다.
- 유형 변환이 필요할 때마다 캐스트 연산자를 사용하는 것은 좋은 프로그래밍 방법으로 간주됩니다.

정수로 끌어올리기

Integer Promotion

- 정수 승격은 int 또는 **unsigned** int 보다 "작은" 정수 유형의 값이 **int** 또는 **unsigned int**로 변환 되는 프로세스 입니다.
- 다음은 int에 문자를 추가하는 예입니다.

```
#import <Foundation/Foundation.h>

int main() {
   int  i = 17;
   char c = 'c';  /* ascii value is 99 */
   int sum;

   sum = i + c;
   NSLog(@"Value of sum : %d\n", sum );

   return 0;
}
```

```
Value of sum : 116
```

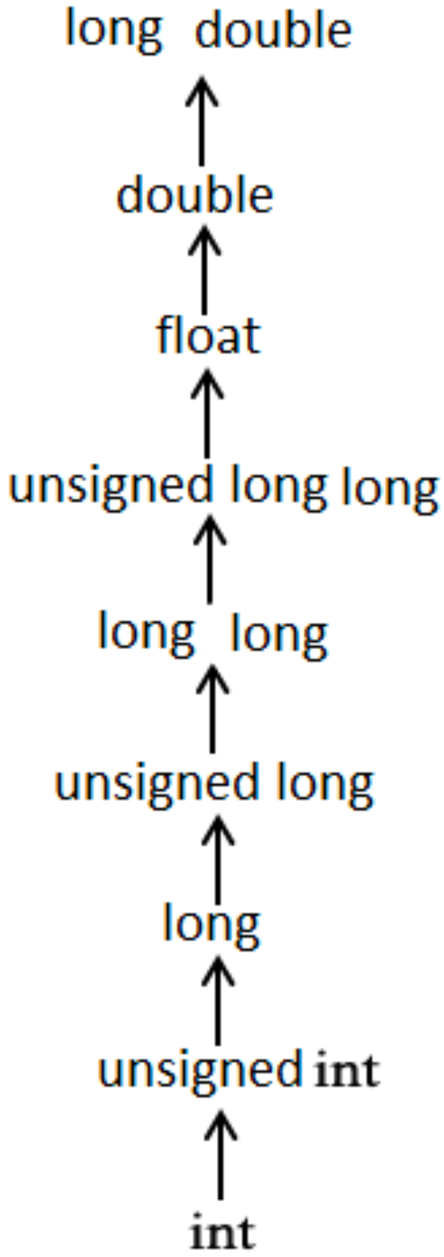
- 여기서 sum 값은 실제 덧셈 연산을 수행하기 전에 컴파일러가 정수 승격을 하고 'c' 값을 ascii로 변환하기 때문에 116 이 됩니다.

일반적인 산술 변환

Usual Arithmetic Conversion

- **일반적인 산술 변환** 은 해당 값을 공통 유형으로 캐스팅하기 위해 암시적으로 수행됩니다.
- 컴파일러는 먼저 정수 승격을 수행합니다.
- 피연산자의 유형이 여전히 다른 경우 다음 계층에서 가장 높은 유형으로 변환 됩니다.

- 할당 연산자나 논리 연산자 && 및 ||에 대해서는 일반적인 산술 변환이 수행되지 않습니다.



일반적인 산술 변환

Usual Arithmetic Conversion

```
#import <Foundation/Foundation.h>

int main() {
   int  i = 17;
   char c = 'c';  /* ascii value is 99 */
   CGFloat sum;

   sum = i + c;
   NSLog(@"Value of sum : %f\n", sum );

   return 0;
}
```

```
Value of sum : 116.000000
```

- 여기서 첫 번째 c는 정수로 변환되지만 최종 값은 부동 소수점이기 때문에 일반적인 산술 변환이 적용되고 컴파일러는 i와 c를 부동 소수점으로 변환하고 더하여 부동 소수점 결과를 산출한다는 것을 이해하는 것은 간단합니다.

로그 처리

NSLog 메서드

NSLog method

- 로그를 인쇄하기 위해 Hello World 예제에서 바로 사용한 Objective-C 프로그래밍 언어의 NSLog 메서드를 사용합니다.
- "Hello World"라는 단어를 인쇄하는 간단한 코드를 살펴보겠습니다.

```
#import <Foundation/Foundation.h>

int main() {
   NSLog(@“Hello, World! \n”);

   return 0;
}
```

```
Hello, World!
```

라이브 앱에서 로그 비활성화

Disabling logs in Live apps

- 우리 앱에서 사용하는 NSLogs 때문에 디바이스의 로그에 출력되고 라이브 빌드에서 로그를 인쇄하는 것은 좋지 않습니다.
- 따라서 로그를 인쇄하기 위해 유형 정의를 사용하고 아래와 같이 사용할 수 있습니다.

```
#import <Foundation/Foundation.h>

#if DEBUG == 0
#define DebugLog(...)
#elif DEBUG == 1
#define DebugLog(...) NSLog(__VA_ARGS__)
#endif

int main() {
   DebugLog(@"Debug log, our custom addition gets \
   printed during debug only" );
   NSLog(@"NSLog gets printed always" );
   return 0;
}
```

[DEBUG MODE]
Debug log, our custom addition gets printed during debug only
NSLog gets printed always

[RELEASE MODE]
NSLog gets printed always

오류 처리

오류 처리

Error Handling

- Objective-C 프로그래밍에서 오류 처리는 Foundation 프레임워크에서 사용할 수 있는 NSError 클래스와 함께 제공됩니다.

- NSError 개체는 오류 코드 또는 오류 문자열만 사용하여 가능한 것보다 더 풍부하고 확장 가능한 오류 정보를 캡슐화합니다.
- NSError 객체의 핵심 속성은 오류 도메인(문자열로 표시), 도메인 특정 오류 코드 및 응용 프로그램 특정 정보를 포함하는 사용자 정보 사전입니다.

NSError

- Objective-C 프로그램은 NSError 객체를 사용하여 사용자에게 알려야 하는 런타임 오류에 대한 정보를 전달합니다.
- 대부분의 경우 프로그램은 이 오류 정보를 대화 상자나 시트에 표시합니다.
- 그러나 정보를 해석하고 사용자에게 오류 복구를 시도하거나 자체적으로 오류 수정을 시도하도록 요청할 수도 있습니다.

- NSError 개체는 다음으로 구성됩니다.
  - **도메인** - 오류 도메인은 미리 정의된 NSError 도메인 중 하나이거나 사용자 정의 도메인을 설명하는 임의의 문자열일 수 있으며 도메인은 nil이 아니어야 합니다.
  - **코드** - 오류에 대한 오류 코드입니다.
  - **사용자 정보** - 오류 및 userInfo에 대한 userInfo 사전은 nil일 수 있습니다.

NSError

- 다음 예는 사용자 지정 오류를 생성하는 방법을 보여줍니다.

```
NSString *domain = @"com.MyCompany.MyApplication.ErrorDomain";
NSString *desc = NSLocalizedString(@"Unable to complete the process", @"");
NSDictionary *userInfo = @{ NSLocalizedDescriptionKey : desc };
NSError *error = [NSError errorWithDomain:domain code:-101 userInfo:userInfo];
```

NSError

```
#import <Foundation/Foundation.h>

@interface SampleClass:NSObject
-(NSString *) getEmployeeNameForID:(int) id withError:(NSError
**)errorPtr;
@end

@implementation SampleClass

-(NSString *) getEmployeeNameForID:(int) id withError:(NSError
**)errorPtr {
   if(id == 1) {
      return @"Employee Test Name";
   } else {
      NSString *domain =
@"com.MyCompany.MyApplication.ErrorDomain";
      NSString *desc =@"Unable to complete the process";
      NSDictionary *userInfo = [[NSDictionary alloc]
      initWithObjectsAndKeys:desc,
      @"NSLocalizedDescriptionKey",NULL];
      *errorPtr = [NSError errorWithDomain:domain code:-101
      userInfo:userInfo];
      return @"";
   }
}

@end
```

```
int main() {
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   SampleClass *sampleClass = [[SampleClass alloc]init];
   NSError *error = nil;
   NSString *name1 = [sampleClass getEmployeeNameForID:1 withError:&error];

   if(error) {
      NSLog(@"Error finding Name1: %@",error);
   } else {
      NSLog(@"Name1: %@",name1);
   }

   error = nil;
   NSString *name2 = [sampleClass getEmployeeNameForID:2 withError:&error];

   if(error) {
      NSLog(@"Error finding Name2: %@",error);
   } else {
      NSLog(@"Name2: %@",name2);
   }

   [pool drain];
   return 0;
}
```

```
Name1: Employee Test Name
Error finding Name2: Unable to complete the process
```

클래스와 객체

클래스와 객체

Classes & Objects

- Objective-C 프로그래밍 언어의 주요 목적은 C 프로그래밍 언어에 객체 지향을 추가하는 것이며 클래스는 객체 지향 프로그래밍을 지원하고 종종 사용자 정의 유형이라고 불리는 Objective-C의 핵심 기능입니다.
- 클래스는 개체의 형식을 지정하는 데 사용되며 데이터 표현과 해당 데이터를 하나의 깔끔한 패키지로 조작하는 방법을 결합합니다. 클래스 내의 데이터와 메소드를 클래스의 멤버라고 합니다.

클래스와 객체

Classes & Objects

- 클래스는 **@interface** 및 **@implementation** 이라는 두 개의 다른 섹션에서 정의됩니다 .
- 거의 모든 것이 객체의 형태입니다.
- 객체는 메시지를 수신하고 객체는 종종 수신자라고 합니다.
- 객체는 인스턴스 변수를 포함합니다.
- 개체 및 인스턴스 변수에는 범위가 있습니다.
- 클래스는 개체의 구현을 숨깁니다.
- 속성은 다른 클래스의 클래스 인스턴스 변수에 대한 액세스를 제공하는 데 사용됩니다.

클래스 정의

Class Definitions

- 클래스를 정의할 때 데이터 유형에 대한 청사진을 정의합니다.
- 이것은 실제로 데이터를 정의하지 않지만 클래스 이름이 의미하는 것, 즉 클래스의 개체가 무엇으로 구성되고 이러한 개체에서 수행할 수 있는 작업을 정의합니다.

- **클래스 정의는 @interface** 키워드와 인터페이스(클래스) 이름으로 시작합니다. 한 쌍의 중괄호로 묶인 클래스 본문. **Objective-C에서 모든 클래스는 NSObject** 라는 기본 클래스에서 파생됩니다 .
- 모든 Objective-C 클래스의 상위 클래스입니다. 메모리 할당 및 초기화와 같은 기본 방법을 제공합니다.
- 예를 들어 다음과 같이 키워드 **class** 를 사용하여 Box 데이터 유형을 정의했습니다.
- 인스턴스 변수는 비공개이며 클래스 구현 내에서만 액세스할 수 있습니다.

```
#import <Foundation/Foundation.h>

@Interface Box:NSObject {
   //Instance variables
   double length;    // Length of a box
   double breadth;   // Breadth of a box
}
@property(nonatomic, readwrite) double height;  // Property

@end
```

개체 할당 및 초기화

Allocating and Initializing Objects

- 클래스는 객체에 대한 청사진을 제공하므로 기본적으로 객체는 클래스에서 생성됩니다. 기본 유형의 변수를 선언하는 것과 정확히 같은 종류의 선언으로 클래스의 개체를 선언합니다. 다음 명령문은 Box 클래스의 두 객체를 선언합니다.
- box1 및 box2 개체 모두 데이터 멤버의 고유한 복사본을 갖습니다.

```
Box box1 = [[Box alloc]init];     // Create box1 object of type Box
Box box2 = [[Box alloc]init];     // Create box2 object of type Box
```

- 클래스의 개체 속성은 직접 멤버 액세스 연산자(.)를 사용하여 액세스할 수 있습니다.

데이터 멤버 액세스

Accessing the Data Members

```
#import <Foundation/Foundation.h>

@interface Box:NSObject {
   double length;    // Length of a box
   double breadth;   // Breadth of a box
   double height;    // Height of a box
}

@property(nonatomic, readwrite) double height;  // Property
-(double) volume;
@end

@implementation Box

@synthesize height;

-(id)init {
   self = [super init];
   length = 1.0;
   breadth = 1.0;
   return self;
}

-(double) volume {
   return length*breadth*height;
}
@end
```

```
int main() {
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   Box *box1 = [[Box alloc]init];    // Create box1 object of type Box
   Box *box2 = [[Box alloc]init];    // Create box2 object of type Box

   double volume = 0.0;              // Store the volume of a box here

   // box 1 specification
   box1.height = 5.0;

   // box 2 specification
   box2.height = 10.0;

   // volume of box 1
   volume = [box1 volume];
   NSLog(@"Volume of Box1 : %f", volume);

   // volume of box 2
   volume = [box2 volume];
   NSLog(@"Volume of Box2 : %f", volume);

   [pool drain];
   return 0;
}
```

```
Volume of Box1 : 5.000000
Volume of Box2 : 10.000000
```

속성

Properties

- 속성은 클래스 외부에서 클래스의 인스턴스 변수에 액세스할 수 있도록 하기 위해 Objective-C에 도입되었습니다.
  - ㅇ속성 은 키워드인 @property 로 시작합니다.
  - 그 뒤에는 비원자 또는 원자, readwrite 또는 readonly 및 strong, unsafe_unretained 또는 weak인 액세스 지정자가 옵니다. 이것은 변수의 유형에 따라 다릅니다. 모든 포인터 유형에 대해 strong, unsafe_unretained 또는 weak를 사용할 수 있습니다. 마찬가지로 다른 유형의 경우 readwrite 또는 readonly를 사용할 수 있습니다.
  - 그 다음에 변수의 데이터 유형이 옵니다.
  - 마지막으로 세미콜론으로 끝나는 속성 이름이 있습니다.
  - 구현 클래스에 synthesize 문을 추가할 수 있습니다. 그러나 최신 XCode에서는 합성 부분을 XCode에서 처리하므로 synthesize 문을 포함할 필요가 없습니다.

속성

Properties

- 클래스의 인스턴스 변수에 액세스할 수 있는 속성에서만 가능합니다. 실제로 속성에 대해 내부적으로 getter 및 setter 메서드가 생성됩니다.
- **예를 들어, @property (nonatomic,readonly) BOOL isDone** 속성이 있다고 가정해 보겠습니다 . 이 예제에는 아래와 같이 생성된 setter와 getter가 있습니다.

```
-(void)setIsDone(BOOL)isDone;
-(BOOL)isDone;
```

상속

상속

Inheritance

- 객체 지향 프로그래밍에서 가장 중요한 개념 중 하나는 상속입니다. 상속을 통해 다른 클래스로 클래스를 정의할 수 있으므로 애플리케이션을 더 쉽게 만들고 유지 관리할 수 있습니다. 이것은 또한 코드 기능과 빠른 구현 시간을 재사용할 수 있는 기회를 제공합니다.
- 클래스를 생성할 때 완전히 새로운 데이터 멤버와 멤버 함수를 작성하는 대신 프로그래머는 새 클래스가 기존 클래스의 멤버를 상속하도록 지정할 수 있습니다. 이 기존 클래스를 **기본** 클래스라고 하고 새 클래스를 **파생** 클래스라고 합니다.
- 상속을 구현한다는 아이디어는 **관계** 입니다. 예를 들어, 포유류 IS-A 동물, 개 IS-A 포유류, 따라서 개 IS-A 동물 등입니다.

기본 및 파생 클래스

Base & Derived Classes

- Objective-C는 다단계 상속만 허용합니다. 즉, 기본 클래스는 하나만 가질 수 있지만 다단계 상속은 허용합니다. **Objective-C의 모든 클래스는 NSObject** 수퍼클래스에서 파생됩니다.

```
@interface derived-class: base-class
```

- 다음과 같이 기본 클래스 Person과 파생 클래스 Employee를 생각해봅시다.

데이터 멤버 액세스

Accessing the Data Members

```
#import <Foundation/Foundation.h>

@interface Person : NSObject {
  NSString *personName;
  NSInteger personAge;
}

- (id)initWithName:(NSString *)name andAge:(NSInteger)age;
- (void)print;

@end

@implementation Person

- (id)initWithName:(NSString *)name andAge:(NSInteger)age {
  personName = name;
  personAge = age;
  return self;
}

- (void)print {
  NSLog(@"Name: %@", personName);
  NSLog(@"Age: %ld", personAge);
}
@end
```

```
@interface Employee : Person {
  NSString *employeeEducation;
}

- (id)initWithName:(NSString *)name andAge:(NSInteger)age
  andEducation:(NSString *)education;
- (void)print;
@end

@implementation Employee

- (id)initWithName:(NSString *)name andAge:(NSInteger)age
  andEducation: (NSString *)education {
    personName = name;
    personAge = age;
    employeeEducation = education;
    return self;
  }

- (void)print {
  NSLog(@"Name: %@", personName);
  NSLog(@"Age: %ld", personAge);
  NSLog(@"Education: %@", employeeEducation);
}
@end
```

데이터 멤버 액세스

Accessing the Data Members

```
int main(int argc, const char * argv[]) {
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSLog(@"Base class Person Object");
   Person *person = [[Person alloc]initWithName:@"Raj" andAge:5];
   [person print];
   NSLog(@"Inherited Class Employee Object");
   Employee *employee = [[Employee alloc]initWithName:@"Raj"
   andAge:5 andEducation:@"MBA"];
   [employee print];
   [pool drain];
   return 0;
}
```

```
Base class Person Object
Name: Raj
Age: 5
Inherited Class Employee Object
Name: Raj
Age: 5
Education: MBA
```

액세스 제어 및 상속

Access Control and Inheritance

- 파생 클래스는 인터페이스 클래스에 정의된 경우 기본 클래스의 모든 private 멤버에 액세스할 수 있지만 구현 파일에 정의된 private 멤버에는 액세스할 수 없습니다.

- 다음과 같은 방식으로 액세스할 수 있는 사람에 따라 다양한 액세스 유형을 요약할 수 있습니다.

- 파생 클래스는 다음 예외를 제외하고 모든 기본 클래스 메서드와 변수를 상속합니다.
  - 확장을 사용하여 구현 파일에 선언된 변수에 액세스할 수 없습니다.
  - 확장을 사용하여 구현 파일에 선언된 메서드에 액세스할 수 없습니다.
  - 상속된 클래스가 기본 클래스의 메서드를 구현하는 경우 파생 클래스의 메서드가 실행됩니다.

조원들과 함께

함께 해요

- 앞서 본 Objective-C로 작성된 다형성 예제 코드를 동일한 역할과 작동이 가능하도록 Swift로 다시 작성해봅시다.
  - 처음엔 단순히 동일한 동작을 하는 코드가 되도록 문법 그대로 옮겨봅시다.
  - Swift의 문법을 적극적으로 사용해 좀 더 사용해 더 나은 코드가 되도록 개선해봅시다.

다형성

다형성

Polymorphism

- **다형성** 이라는 단어 는 많은 형태를 갖는다는 의미입니다. 일반적으로 다형성은 클래스의 계층 구조가 있고 상속으로 관련되어 있을 때 발생합니다.
- Objective-C 다형성은 멤버 함수에 대한 호출이 함수를 호출하는 객체의 유형에 따라 다른 함수가 실행되도록 한다는 것을 의미합니다.
- 예를 들어, 모든 모양에 대한 기본 인터페이스를 제공하는 Shape 클래스가 있습니다. Square 및 Rectangle은 기본 클래스인 Shape에서 파생됩니다.

다형성

Polymorphism

```
#import <Foundation/Foundation.h>

@interface Shape : NSObject {
  CGFloat area;
}

- (void)printArea;
- (void)calculateArea;
@end

@implementation Shape
- (void)printArea {
  NSLog(@"The area is %f", area);
}

- (void)calculateArea {

}

@end
```

```
@interface Square : Shape {
  CGFloat length;
}

- (id)initWithSide:(CGFloat)side;
- (void)calculateArea;
@end

@implementation Square
- (id)initWithSide:(CGFloat)side {
  length = side;
  return self;
}

- (void)calculateArea {
  area = length * length;
}

- (void)printArea {
  NSLog(@"The area of square is %f", area);
}
@end
```

```
@interface Rectangle : Shape {
  CGFloat length;
  CGFloat breadth;
}

- (id)initWithLength:(CGFloat)rLength andBreadth:(CGFloat)rBreadth;
@end

@implementation Rectangle
- (id)initWithLength:(CGFloat)rLength andBreadth:(CGFloat)rBreadth {
  length = rLength;
  breadth = rBreadth;
  return self;
}

- (void)calculateArea {
  area = length * breadth;
}
@end
```

다형성

Polymorphism

```
int main(int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    Shape *square = [[Square alloc]initWithSide:10.0];
    [square calculateArea];
    [square printArea];

    Shape *rect = [[Rectangle alloc]
    initWithLength:10.0 andBreadth:5.0];
    [rect calculateArea];
    [rect printArea];

    [pool drain];
    return 0;
}
```

```
The area of square is 100.000000
The area is 50.000000
```

- 위의 예제에서는 computeArea 및 printArea 메서드의 가용성을 기반으로 기본 클래스 또는 파생 클래스의 메서드가 실행되었습니다.
- 다형성은 두 클래스의 메서드 구현을 기반으로 기본 클래스와 파생 클래스 간의 메서드 전환을 처리합니다.

조원들과 함께

함께 해요

- 앞서 본 Objective-C로 작성된 다형성 예제 코드를 동일한 역할과 작동이 가능하도록 Swift로 다시 작성해봅시다.
  - 처음엔 단순히 동일한 동작을 하는 코드가 되도록 문법 그대로 옮겨봅시다.
  - Swift의 문법을 적극적으로 사용해 좀 더 사용해 더 나은 코드가 되도록 개선해봅시다.

데이터 캡슐화

데이터 캡슐화

Data Encapsulation

- 모든 Objective-C 프로그램은 다음 두 가지 기본 요소로 구성됩니다.
  - **프로그램 문(코드)** - 이것은 작업을 수행하는 프로그램의 일부이며 이를 메서드라고 합니다.
  - **프로그램 데이터** - 데이터는 프로그램 기능의 영향을 받는 프로그램 정보입니다.

- 캡슐화는 데이터와 데이터를 조작하는 기능을 함께 묶고 외부 간섭과 오용으로부터 안전하게 유지하는 객체 지향 프로그래밍 개념입니다.
- **데이터 캡슐화는 데이터 은닉** 이라는 중요한 OOP 개념으로 이어졌습니다 .

- **데이터 캡슐화** 는 데이터와 이를 사용하는 기능을 묶는 메커니즘이고 **데이터 추상화** 는 인터페이스만 노출하고 구현 세부 사항을 사용자에게 숨기는 메커니즘입니다.

데이터 캡슐화

Data Encapsulation

- Objective-C는 **클래스**라고 하는 사용자 정의 유형의 생성을 통해 캡슐화 및 데이터 은닉의 속성을 지원합니다.

```
@interface Adder : NSObject {
  NSInteger total;
}

- (id)initWithInitialNumber:(NSInteger)initialNumber;
- (void)addNumber:(NSInteger)newNumber;
- (NSInteger)getTotal;

@end
```

- 변수 total은 비공개이며 클래스 외부에서 액세스할 수 없습니다. 이것은 Adder 클래스의 다른 멤버만 액세스할 수 있으며 프로그램의 다른 부분에서는 액세스할 수 없음을 의미합니다. 이것은 캡슐화를 달성하는 한 가지 방법입니다.
- 인터페이스 파일 내의 메서드는 액세스할 수 있으며 범위 내에서 공개됩니다.
- 다음 장에서 배우게 될 **확장 기능**(extension)도 참고하세요.

데이터 캡슐화 예

Data Encapsulation Example

- public 및 private 멤버 변수를 사용하여 클래스를 구현하는 모든 Objective-C 프로그램은 데이터 캡슐화 및 데이터 추상화의 예입니다.

```
#import <Foundation/Foundation.h>

@interface Adder : NSObject {
   NSInteger total;
}

- (id)initWithInitialNumber:(NSInteger)initialNumber;
- (void)addNumber:(NSInteger)newNumber;
- (NSInteger)getTotal;
@end
```

```
@implementation Adder
-(id)initWithInitialNumber:(NSInteger)initialNumber {
   total = initialNumber;
   return self;
}

- (void)addNumber:(NSInteger)newNumber {
   total = total + newNumber;
}

- (NSInteger)getTotal {
   return total;
}
@end
```

데이터 캡슐화 예

Data Encapsulation Example

```
int main(int argc, const char * argv[]) {
    NSAutoreleasePool * pool
        = [[NSAutoreleasePool alloc] init];

    Adder *adder = [[Adder alloc]initWithInitialNumber:10];
    [adder addNumber:5];
    [adder addNumber:4];

    NSLog(@"The total is %ld",[adder getTotal]);

    [pool drain];
    return 0;
}
```

```
The total is 19
```

- 위의 클래스는 숫자를 더하고 합계를 반환합니다.
- public 멤버인 **addNum** 및 **getTotal** 은 외부 세계에 대한 인터페이스이며 사용자는 클래스를 사용하기 위해 이를 알아야 합니다.
- private 멤버 **total** 은 외부 세계에 숨겨져 있지만 클래스가 제대로 작동하기 위해 필요한 것입니다.

전략 설계

Designing Strategy

- 우리 대부분은 실제로 노출해야 하는 경우가 아니면 기본적으로 클래스 멤버를 비공개로 설정하는 쓰라린 경험을 통해 배웠습니다. 이것이 좋은 **캡슐화** 입니다.
- 데이터 캡슐화는 Objective-C를 포함한 모든 객체 지향 프로그래밍(OOP) 언어의 핵심 기능 중 하나이기 때문에 데이터 캡슐화를 이해하는 것이 중요합니다.

카테고리

카테고리

Categories

- 때로는 특정 상황에서만 유용한 동작을 추가하여 기존 클래스를 확장하려는 경우가 있습니다. 이러한 확장을 기존 클래스에 추가하기 위해 Objective-C는 **범주(카테고리)** 및 **확장(익스텐션)** 을 제공합니다 .
- 기존 클래스에 메서드를 추가해야 하는 경우, 아마도 자신의 애플리케이션에서 더 쉽게 수행할 수 있도록 기능을 추가해야 하는 경우 가장 쉬운 방법은 카테고리를 사용하는 것입니다.
- 카테고리를 선언하는 구문은 표준 Objective-C 클래스 설명과 마찬가지로 @interface 키워드를 사용하지만 하위 클래스로부터의 상속을 나타내지는 않습니다. 대신, 다음과 같이 괄호 안에 카테고리 이름을 지정합니다.

```
@interface ClassName (CategoryName)

@end
```

카테고리의 특성

Characteristics of Category

- 원래 구현 소스 코드가 없더라도 모든 클래스에 대해 카테고리를 선언할 수 있습니다.
- 카테고리에서 선언하는 모든 메서드는 원래 클래스의 모든 하위 클래스뿐만 아니라 원래 클래스의 모든 인스턴스에서 사용할 수 있습니다.
- 런타임에는 카테고리에 의해 추가된 메서드와 원래 클래스에 의해 구현되는 메서드 사이에 차이가 없습니다.

카테고리의 특성

Characteristics of Category

- Cocoa 클래스 NSString에 카테고리를 추가해 보겠습니다.
- 이 카테고리를 사용하면 저작권 문자열을 반환하는 데 도움이 되는 새 메서드 getCopyRightString을 추가할 수 있습니다.

```
#import <Foundation/Foundation.h>

@interface NSString(MyAdditions)
+(NSString *)getCopyRightString;
@end

@implementation NSString(MyAdditions)
+(NSString *)getCopyRightString {
   return @"Copyright TutorialsPoint.com 2013";
}
@end
```

```
int main(int argc, const char * argv[]) {
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSString *copyrightString = [NSString getCopyRightString];
   NSLog(@"Accessing Category: %@",copyrightString);

   [pool drain];
   return 0;
}
```

```
Accessing Category: Copyright TutorialsPoint.com 2013
```

카테고리의 특성

Characteristics of Category

- 카테고리에 의해 추가된 메서드는 클래스 및 해당 하위 클래스의 모든 인스턴스에서 사용할 수 있지만 추가 메서드를 사용하려는 소스 코드 파일에서 범주 헤더 파일을 가져와야 합니다. 그렇지 않으면 컴파일러 경고 및 오류.
- 이 예에서는 클래스가 하나뿐이므로 헤더 파일을 포함하지 않았으므로 위와 같은 경우 헤더 파일을 포함해야 합니다.

확장

확장

Extensions

- 클래스 확장은 카테고리와 어느 정도 유사하지만 컴파일 시간에 소스 코드가 있는 클래스에만 추가할 수 있습니다(클래스는 클래스 확장과 동시에 컴파일됨).
- 클래스 확장으로 선언된 메서드는 원래 클래스의 구현 블록에서 구현되므로 예를 들어 NSString과 같은 Cocoa 또는 Cocoa Touch 클래스와 같은 프레임워크 클래스에서 클래스 확장을 선언할 수 없습니다.
- 확장은 실제로 범주 이름이 없는 범주입니다. 종종 **익명 카테고리** 라고 합니다.
- 확장을 선언하는 구문은 표준 Objective-C 클래스 설명과 마찬가지로 @interface 키워드를 사용하지만 하위 클래스로부터의 상속을 나타내지는 않습니다. 대신 아래와 같이 괄호만 추가합니다.

```
@interface ClassName ( )

@end
```

확장의 특성

Characteristics of Extensions

- 확장은 모든 클래스에 대해 선언할 수 없으며 소스 코드의 원래 구현이 있는 클래스에만 적용됩니다.
- 확장은 해당 클래스에만 해당하는 개인 메서드와 개인 변수를 추가하는 것입니다.
- 확장 내부에 선언된 메서드나 변수는 상속된 클래스에서도 액세스할 수 없습니다.

확장 예

Extensions Example

- 확장이 있는 SampleClass 클래스를 생성해 보겠습니다. 확장에 private 변수 internalID가 있습니다.
- 그럼 내부ID를 처리한 후 외부ID를 반환하는 getExternalID 메소드를 만들어보자.
- 예제는 아래에 나와 있으며 온라인 컴파일러에서는 작동하지 않습니다.

```
#import <Foundation/Foundation.h>

@interface SampleClass : NSObject {
   NSString *name;
}

- (void)setInternalID;
- (NSString *)getExternalID;

@end

@interface SampleClass() {
   NSString *internalID;
}
@end
```

```
@implementation SampleClass

- (void)setInternalID {
   internalID = [NSString stringWithFormat:
   @"UNIQUEINTERNALKEY%dUNIQUEINTERNALKEY",arc4random()%100];
}

- (NSString *)getExternalID {
   return [internalID stringByReplacingOccurrencesOfString:
   @"UNIQUEINTERNALKEY" withString:@""];
}

@end
```

확장 예

Extensions Example

```
int main(int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    SampleClass *sampleClass = [[SampleClass alloc]init];
    [sampleClass setInternalID];
    NSLog(@"ExternalID: %@",[sampleClass getExternalID]);
    [pool drain];
    return 0;
}
```

```
ExternalID: 51
```

- 위의 예에서 internalID가 직접 반환되지 않는 것을 볼 수 있습니다. 여기서 UNIQUEINTERNALKEY를 제거하고 나머지 값만 getExternalID 메서드에서 사용할 수 있도록 합니다.
- 위의 예는 문자열 연산만을 사용하지만 암호화/복호화 등 많은 기능을 가질 수 있습니다.

프로토콜

프로토콜

Protocols

- Objective-C를 사용하면 특정 상황에 사용될 것으로 예상되는 메서드를 선언하는 프로토콜을 정의할 수 있습니다. 프로토콜은 프로토콜을 준수하는 클래스에서 구현됩니다.
- 간단한 예는 네트워크 URL 처리 클래스이며, 네트워크 URL 가져오기 작업이 끝나면 호출 클래스에 친밀감을 주는 processCompleted 대리자 메서드와 같은 메서드가 있는 프로토콜이 있습니다.
- 프로토콜의 구문은 다음과 같습니다.

```
@protocol ProtocolName
@required
    // list of required methods
@optional
// list of optional methods
@end
```

프로토콜

Protocols

- Objective-C를 사용하면 특정 상황에 사용될 것으로 예상되는 메서드를 선언하는 프로토콜을 정의할 수 있습니다. 프로토콜은 프로토콜을 준수하는 클래스에서 구현됩니다.
- 간단한 예는 네트워크 URL 처리 클래스이며, 네트워크 URL 가져오기 작업이 끝나면 호출 클래스에 친밀감을 주는 processCompleted 대리자 메서드와 같은 메서드가 있는 프로토콜이 있습니다.
- 프로토콜의 구문은 다음과 같습니다.

```
@protocol ProtocolName
@required
    // list of required methods
@optional
// list of optional methods
@end
```

- **@required** 키워드 아래의 메서드는 프로토콜을 준수하는 클래스에서 구현되어야 하며 **@optional** 키워드 아래의 메서드는 선택적으로 구현해야 합니다.

프로토콜

Protocols

- 다음은 프로토콜을 준수하는 클래스의 구문입니다.

```
@interface MyClass : NSObject <MyProtocol>
...
@end
```

- 이것은 MyClass의 모든 인스턴스가 인터페이스에서 구체적으로 선언된 메소드에 응답할 뿐만 아니라 MyClass가 MyProtocol의 필수 메소드에 대한 구현도 제공한다는 것을 의미합니다.
- 클래스 인터페이스에서 프로토콜 메서드를 다시 선언할 필요가 없습니다. 프로토콜을 채택하면 충분합니다.

- 여러 프로토콜을 채택하는 클래스가 필요한 경우 쉼표로 구분된 목록으로 지정할 수 있습니다.
- 프로토콜을 구현하는 호출 개체의 참조를 보유하는 대리자 개체가 있습니다.

프로토콜

Protocols

```
#import <Foundation/Foundation.h>

@protocol PrintProtocolDelegate
- (void)processCompleted;

@end

@interface PrintClass :NSObject {
   id delegate;
}

- (void) printDetails;
- (void) setDelegate:(id)newDelegate;
@end

@implementation PrintClass
- (void)printDetails {
   NSLog(@"Printing Details");
   [delegate processCompleted];
}

- (void) setDelegate:(id)newDelegate {
   delegate = newDelegate;
}
@end
```

```
@interface SampleClass:NSObject<PrintProtocolDelegate>
- (void)startAction;

@end

@implementation SampleClass
- (void)startAction {
   PrintClass *printClass = [[PrintClass alloc]init];
   [printClass setDelegate:self];
   [printClass printDetails];
}

-(void)processCompleted {
   NSLog(@"Printing Process Completed");
}

@end

int main(int argc, const char * argv[]) {
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   SampleClass *sampleClass = [[SampleClass alloc]init];
   [sampleClass startAction];
   [pool drain];
   return 0;
}
```

```
Printing Details
Printing Process Completed
```

프로토콜

Protocols

- 앞의 예에서 우리는 delgate 메소드가 어떻게 호출되고 실행되는지 보았습니다.
- startAction으로 시작하여 프로세스가 완료되면 대리자 메서드인 processCompleted가 호출되어 작업이 완료되었음을 알립니다.

- 모든 iOS 또는 Mac 앱에서는 대리인 없이 프로그램을 구현하지 않습니다.
- 그래서 우리는 대리자의 사용법을 이해하는 것이 중요합니다.
- 대리자 개체는 메모리 누수를 방지하기 위해 unsafe_unretained 속성 유형을 사용해야 합니다.

동적 바인딩

동적 바인딩

Dynamic Binding

- 동적 바인딩은 컴파일 시간이 아닌 런타임에 호출할 메서드를 결정합니다.
- 동적 바인딩은 후기 바인딩이라고도 합니다.

- Objective-C에서 모든 메소드는 런타임에 동적으로 해결됩니다.
- 실행되는 정확한 코드는 메소드 이름(선택자)과 수신 객체에 의해 결정됩니다.

- 동적 바인딩은 다형성을 가능하게 합니다.
- 예를 들어, Rectangle 및 Square를 포함한 개체 컬렉션을 고려하십시오.
- 각 개체에는 고유한 printArea 메서드 구현이 있습니다.

- 다음 코드에서 [anObject printArea] 표현식에 의해 실행되어야 하는 실제 코드는 런타임에 결정됩니다.
- 런타임 시스템은 메서드 실행을 위한 선택기를 사용하여 Object의 클래스가 무엇이든 적절한 메서드를 식별합니다.

동적 바인딩

Dynamic Binding

```
#import <Foundation/Foundation.h>

@interface Square:NSObject {
   float area;
}

- (void)calculateAreaOfSide:(CGFloat)side;
- (void)printArea;
@end

@implementation Square
- (void)calculateAreaOfSide:(CGFloat)side {
   area = side * side;
}

- (void)printArea {
   NSLog(@"The area of square is %f",area);
}
@end
```

```
@interface Rectangle:NSObject {
   float area;
}

- (void)calculateAreaOfLength:(CGFloat)length andBreadth:
(CGFloat)breadth;
- (void)printArea;
@end

@implementation  Rectangle

- (void)calculateAreaOfLength:(CGFloat)length andBreadth:
(CGFloat)breadth {
   area = length * breadth;
}

- (void)printArea {
   NSLog(@"The area of Rectangle is %f",area);
}
@end
```

동적 바인딩

Dynamic Binding

```
int main() {
   Square *square = [[Square alloc]init];
   [square calculateAreaOfSide:10.0];

   Rectangle *rectangle = [[Rectangle alloc]init];
   [rectangle calculateAreaOfLength:10.0 andBreadth:5.0];

   NSArray *shapes = [[NSArray alloc]initWithObjects: square, rectangle,nil];
   id object1 = [shapes objectAtIndex:0];
   [object1 printArea];

   id object2 = [shapes objectAtIndex:1];
   [object2 printArea];

   return 0;
}
```

```
The area of square is 100.000000
The area of Rectangle is 50.000000
```

- 위의 예에서 볼 수 있듯이 printArea 메서드는 런타임에 동적으로 선택됩니다.
- 이는 동적 바인딩의 예이며 유사한 종류의 개체를 처리할 때 많은 상황에서 매우 유용합니다.

복합 객체

복합 객체

Composite Objects

- 클래스 클러스터 내에 객체를 포함하는 클래스를 정의하는 하위 클래스를 생성할 수 있습니다.
- 이러한 클래스 개체는 복합 객체입니다.

- 따라서 클래스 클러스터가 무엇인지 궁금할 수 있습니다. 따라서 먼저 클래스 클러스터가 무엇인지 살펴보겠습니다.

클래스 클러스터

Class Clusters

- 클래스 클러스터는 Foundation 프레임워크에서 광범위하게 사용하는 디자인 패턴입니다.
- 클래스 클러스터는 공개 추상 수퍼클래스 아래에 여러 개인 구체적인 하위 클래스를 그룹화합니다.
- 이러한 방식으로 클래스를 그룹화하면 기능적 풍부함을 줄이지 않으면서 객체 지향 프레임워크의 공개적으로 보이는 아키텍처를 단순화할 수 있습니다.
- 클래스 클러스터는 Abstract Factory 디자인 패턴을 기반으로 합니다.

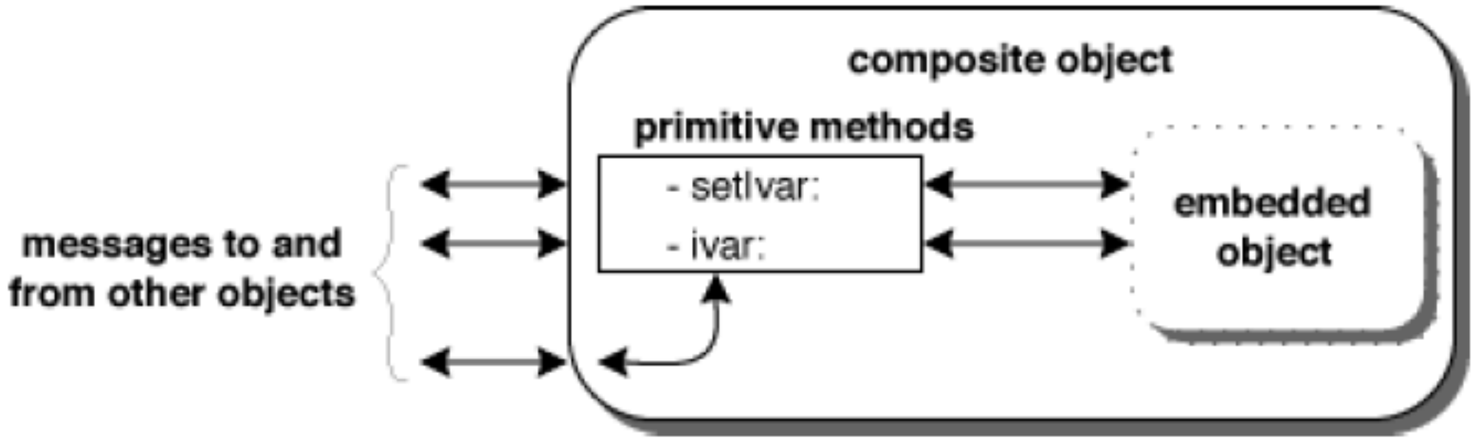
- 간단하게 하기 위해 유사한 기능에 대해 여러 클래스를 만드는 대신 입력 값을 기반으로 처리를 처리하는 단일 클래스를 만듭니다.

- 예를 들어 NSNumber에는 char, int, bool 등과 같은 클래스 클러스터가 많이 있습니다.
- 단일 클래스에서 유사한 작업을 처리하는 단일 클래스로 모든 것을 그룹화합니다.
- NSNumber는 실제로 이러한 기본 유형의 값을 개체로 래핑합니다.

그렇다면 복합 객체란 정확히 무엇일까요?

What is a Composite Object?

- 자체 설계한 개체에 개인 클러스터 개체를 포함하여 복합 개체를 만듭니다.
- 이 복합 개체는 기본 기능을 위해 클러스터 개체에 의존할 수 있으며 복합 개체가 특정 방식으로 처리하려는 메시지만 가로채게 됩니다.
- 이 아키텍처는 작성해야 하는 코드의 양을 줄이고 Foundation Framework에서 제공하는 테스트된 코드를 활용할 수 있도록 합니다.



Courtesy: Apple Documentation

그렇다면 복합 객체란 정확히 무엇일까요?

What is a Composite Object?

- 복합 개체는 클러스터의 추상 슈퍼클래스의 하위 클래스로 자신을 선언해야 합니다.
- 하위 클래스로서 상위 클래스의 기본 메서드를 재정의해야 합니다.
- 파생 메서드를 재정의할 수도 있지만 파생 메서드는 기본 메서드를 통해 작동하기 때문에 필요하지 않습니다.

- NSArray 클래스의 count 메소드가 한 예입니다. 재정의하는 메서드의 중간 개체 구현은 다음과 같이 간단할 수 있습니다.
- 예에서 포함된 객체는 실제로 NSArray 유형입니다.

```
- (unsigned)count  {
   return [embeddedObject count];
}
```

복합 객체 예제

A Composite Object Example

```
#import <Foundation/Foundation.h>

@interface ValidatingArray : NSMutableArray {
  NSMutableArray *embeddedArray;
}

+ validatingArray;
- init;
- (unsigned)count;
- objectAtIndex:(unsigned)index;
- (void)addObject:object;
- (void)replaceObjectAtIndex:(unsigned)index withObject:object;
- (void)removeLastObject;
- (void)insertObject:object atIndex:(unsigned)index;
- (void)removeObjectAtIndex:(unsigned)index;

@end
```

```
@implementation ValidatingArray
- init {
  self = [super init];
  if (self) {
    embeddedArray = [[NSMutableArray allocWithZone:[self zone]] init];
  }
  return self;
}

+ validatingArray {
  return [[self alloc] init] ;
}

- (unsigned)count {
  return [embeddedArray count];
}

- objectAtIndex:(unsigned)index {
  return [embeddedArray objectAtIndex:index];
}
```

복합 객체 예제

A Composite Object Example

```
- (void)addObject:(id)object {
   if (object != nil) {
      [embeddedArray addObject:object];
   }
}

- (void)replaceObjectAtIndex:(unsigned)index withObject:(id)object; {
   if (index <[embeddedArray count] && object != nil) {
      [embeddedArray replaceObjectAtIndex:index withObject:object];
   }
}

- (void)removeLastObject; {
   if ([embeddedArray count] > 0) {
      [embeddedArray removeLastObject];
   }
}

- (void)insertObject:(id)object atIndex:(unsigned)index; {
   if (object != nil) {
      [embeddedArray insertObject:object atIndex:index];
   }
}
```

```
- (void)removeObjectAtIndex:(unsigned)index; {
   if (index <[embeddedArray count]) {
      [embeddedArray removeObjectAtIndex:index];
   }
}
@end

int main() {
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   ValidatingArray *validatingArray = [ValidatingArray validatingArray];

   [validatingArray addObject:@"Object1"];
   [validatingArray addObject:@"Object2"];
   [validatingArray addObject:[NSNull null]];
   [validatingArray removeObjectAtIndex:2];
   NSString *aString = [validatingArray objectAtIndex:1];
   NSLog(@"The value at Index 1 is %@",aString);
   [pool drain];

   return 0;
}
```

```
The value at Index 1 is Object2
```

복합 객체 예제

A Composite Object Example

- 앞의 예에서 배열의 하나의 기능을 검증하면 정상적인 시나리오에서 충돌을 일으킬 null 개체를 추가할 수 없음을 알 수 있습니다.
- 그러나 우리의 검증 배열이 그것을 처리합니다.
- 마찬가지로 배열 유효성 검사의 각 방법은 일반적인 작업 순서와 별도로 유효성 검사 프로세스를 추가합니다.

Foundation Framework

Foundation Framework

- Apple 문서를 참조하면 아래와 같이 Foundation 프레임워크의 내용을 확인할 수 있습니다.
- Foundation 프레임워크는 Objective-C 클래스의 기본 계층을 정의합니다. 유용한 기본 객체 클래스 세트를 제공하는 것 외에도 Objective-C 언어에서 다루지 않는 기능을 정의하는 여러 패러다임을 소개합니다. 재단 프레임워크는 이러한 목표를 염두에 두고 설계되었습니다.
  - 작은 기본 유틸리티 클래스 세트를 제공합니다.
  - 할당 해제와 같은 일관된 규칙을 도입하여 소프트웨어 개발을 더 쉽게 만듭니다.
  - 유니코드 문자열, 개체 지속성 및 개체 배포를 지원합니다.
  - 이식성 향상을 위해 OS 독립성 수준을 제공합니다.
- 프레임워크는 Apple이 인수한 NeXTStep에서 개발했으며 이러한 기초 클래스는 Mac OS X 및 iOS의 일부가 되었습니다.
- NeXTStep에 의해 개발되었기 때문에 클래스 접두사 "NS"가 있습니다.

Foundation Framework

- 우리는 모든 샘플 프로그램에서 Foundation Framework를 사용했습니다. Foundation Framework를 사용하는 것은 거의 필수입니다.

- **일반적으로 #import <Foundation/NSString.h>** 와 같은 것을 사용 하여 Objective-C 클래스를 가져오지만 너무 많은 클래스를 가져오는 것을 방지하기 위해 모두 **#import <Foundation/Foundation.h>** 에서 가져옵니다 .

- NSObject는 기초 키트 클래스를 포함한 모든 개체의 기본 클래스입니다.
- 메모리 관리 방법을 제공합니다.
- 또한 런타임 시스템에 대한 기본 인터페이스와 Objective-C 개체로 동작하는 기능을 제공합니다.
- 기본 클래스가 없으며 모든 클래스의 루트입니다.

기능별 Foundation의 클래스들

Foundation Classes based on functionality

	설명
데이터 저장소 Data storage	NSArray, NSDictionary 및 NSSet은 모든 클래스의 Objective-C 개체에 대한 저장소를 제공합니다.
텍스트와 문자열 Text and strings	NSCharacterSet은 NSString 및 NSScanner 클래스에서 사용되는 다양한 문자 그룹을 나타냅니다. NSString 클래스는 텍스트 문자열을 나타내며 문자열 검색, 결합 및 비교를 위한 메서드를 제공합니다. NSScanner 개체는 NSString 개체에서 숫자와 단어를 검색하는 데 사용됩니다.
날짜와 시간 Dates and times	NSDate, NSTimeZone 및 NSCalendar 클래스는 시간과 날짜를 저장하고 달력 정보를 나타냅니다. 날짜 및 시간 차이를 계산하는 방법을 제공합니다. NSLocale과 함께 날짜와 시간을 다양한 형식으로 표시하고 세계의 위치에 따라 시간과 날짜를 조정하는 방법을 제공합니다.
예외 처리 Exception handling	예외 처리는 예기치 않은 상황을 처리하는 데 사용되며 NSException과 함께 Objective-C에서 제공됩니다.
파일 처리 File handling	파일 처리는 NSFileManager 클래스의 도움으로 수행됩니다.
URL 로딩 시스템 URL loading system	공통 인터넷 프로토콜에 대한 액세스를 제공하는 클래스 및 프로토콜 집합입니다.

데이터 저장소

Data storage

- 데이터 저장 및 검색은 모든 프로그램에서 가장 중요한 것 중 하나입니다.
- Objective-C에서는 일반적으로 작업을 복잡하게 만들기 때문에 연결 목록과 같은 구조에 의존하지 않습니다.
- 대신 NSArray, NSSet, NSDictionary 및 변경 가능한 형식과 같은 컬렉션을 사용합니다.

NSArray와 NSMutableArray

Data storage

- NSArray는 변경할 수 없는 객체 배열을 유지하는 데 사용되며 NSMutableArray는 변경 가능한 개체 배열을 유지하는 데 사용됩니다.
- 가변성은 런타임에 미리 할당된 배열을 변경하는 데 도움이 되지만 NSArray를 사용하는 경우 기존 배열만 교체하고 기존 배열의 내용은 변경할 수 없습니다.
- NSArray의 중요한 메소드는 다음과 같습니다.
  - **alloc/initWithObjects** - 객체로 배열을 초기화하는 데 사용됩니다.
  - **objectAtIndex** - 특정 인덱스에 있는 개체를 반환합니다.
  - **count** - 객체의 수를 반환합니다.

NSArray와 NSMutableArray

Data storage

- NSMutableArray는 NSArray에서 상속되므로 NSArray의 모든 인스턴스 메서드는 NSMutableArray에서 사용할 수 있습니다.
- NSMutableArray의 중요한 메소드는 다음과 같습니다.
  - removeAllObjects - 배열을 비웁니다.
  - **addObject** - 배열의 끝에 주어진 객체를 삽입합니다.
  - **removeObjectAtIndex** - 특정 인덱스에서 objectA를 제거하는 데 사용됩니다.
  - exchangeObjectAtIndex:withObjectAtIndex - 주어진 인덱스에서 배열의 개체를 교환합니다.
  - replaceObjectAtIndex:withObject - 인덱스의 개체를 개체로 바꿉니다.

NSArray와 NSMutableArray

Data storage

```
#import <Foundation/Foundation.h>

int main() {
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSArray *array = [[NSArray alloc]
   initWithObjects:@"string1", @"string2",@"string3",nil];
   NSString *string1 = [array objectAtIndex:0];
   NSLog(@"The object in array at Index 0 is %@",string1);

   NSMutableArray *mutableArray = [[NSMutableArray alloc]init];
   [mutableArray addObject: @"string"];
   string1 = [mutableArray objectAtIndex:0];
   NSLog(@"The object in mutableArray at Index 0 is %@",string1);

   [pool drain];
   return 0;
}
```

```
The object in array at Index 0 is string1
The object in mutableArray at Index 0 is string
```

NSDictionary와 NSMutableDictionary

Data storage

- NSDictionary는 개체의 변경 불가능한 사전을 유지하는 데 사용되며 NSMutableDictionary는 개체의 변경 가능한 사전을 유지하는 데 사용됩니다.
- NSDictionary의 중요한 메소드는 다음과 같습니다.
  - **alloc/initWithObjectsAndKeys** - 지정된 값 및 키 세트로 구성된 항목으로 새로 할당된 사전을 초기화합니다.
  - **valueForKey** - 주어진 키와 관련된 값을 반환합니다.
  - **count** - 사전의 항목 수를 반환합니다.

NSDictionary와 NSMutableDictionary

Data storage

- NSMutableDictionary는 NSDictionary에서 상속되므로 NSDictionary의 모든 인스턴스 메서드는 NSMutableDictionary에서 사용할 수 있습니다.
- NSmutableDictionary의 중요한 메소드는 다음과 같습니다.
  - **removeAllObjects** - 항목의 사전을 비웁니다.
  - **removeObjectForKey** - 사전에서 주어진 키와 관련 값을 제거합니다.
  - **setValue:forKey** - 주어진 키-값 쌍을 사전에 추가합니다.

NSDictionary와 NSMutableDictionary

Data storage

```
#import <Foundation/Foundation.h>

int main() {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSDictionary *dictionary = [[NSDictionary alloc] initWithObjectsAndKeys:
    @"string1",@"key1", @"string2",@"key2",@"string3",@"key3",nil];
    NSString *string1 = [dictionary objectForKey:@"key1"];
    NSLog(@"The object for key, key1 in dictionary is %@",string1);

    NSMutableDictionary *mutableDictionary = [[NSMutableDictionary alloc]init];
    [mutableDictionary setValue:@"string" forKey:@"key1"];
    string1 = [mutableDictionary objectForKey:@"key1"];
    NSLog(@"The object for key, key1 in mutableDictionary is %@",string1);

    [pool drain];
    return 0;
}
```

```
The object for key, key1 in dictionary is string1
The object for key, key1 in mutableDictionary is string
```

NSSet와 NSMutableSet

Data storage

- NSSet은 고유한 개체의 변경 불가능한 집합을 유지하는 데 사용되며 NSMutableDictionary는 고유한 개체의 변경 가능한 집합을 유지하는 데 사용됩니다.
- NSSet의 중요한 메소드는 다음과 같습니다.
  - **alloc/initWithObjects** - 지정된 개체 목록에서 가져온 구성원으로 새로 할당된 집합을 초기화합니다.
  - **allObjects** - 집합의 구성원을 포함하는 배열을 반환하거나 집합에 구성원이 없는 경우 빈 배열을 반환합니다.
  - **count** - 집합의 구성원 수를 반환합니다.

NSSet와 NSMutableSet

Data storage

- NSMutableSet은 NSSet에서 상속되므로 NSSet의 모든 인스턴스 메서드는 NSMutableSet에서 사용할 수 있습니다.
- NSMutableSet의 중요한 메소드는 다음과 같습니다.
  - **removeAllObjects** - 모든 구성원 집합을 비웁니다.
  - **addObject** - 이미 구성원이 아닌 경우 지정된 개체를 집합에 추가합니다.
  - **removeObject** - 세트에서 주어진 객체를 제거합니다.

NSSet와 NSMutableSet

Data storage

```
#import <Foundation/Foundation.h>

int main() {
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSSet *set = [[NSSet alloc]
   initWithObjects:@"string1", @"string2",@"string3",nil];
   NSArray *setArray = [set allObjects];
   NSLog(@"The objects in set are %@",setArray);

   NSMutableSet *mutableSet = [[NSMutableSet alloc]init];
   [mutableSet addObject:@"string1"];
   setArray = [mutableSet allObjects];
   NSLog(@"The objects in mutableSet are %@",setArray);

   [pool drain];
   return 0;
}
```

```
The objects in set are (string3, string2, string1)
The objects in mutableSet are (string1)
```

텍스트와 문자열

Text and strings

- NSString은 문자열과 텍스트를 저장하는 데 사용되는 가장 일반적으로 사용되는 클래스 중 하나입니다.
- NSString에 대해 더 알고 싶다면 Objective-C strings에서 NSString을 참조하세요.

- 앞서 언급했듯이 NSCharacterSet은 NSString 및 NSScanner 클래스에서 사용되는 다양한 문자 그룹을 나타냅니다.

NSCharacterSet

Text and strings

- 다음은 다양한 문자 집합을 나타내는 NSCharacterSet에서 사용할 수 있는 메서드 집합입니다.
  - **alphanumericCharacterSet** - 문자, 표시 및 숫자 범주의 문자를 포함하는 문자 집합을 반환합니다.
  - **CapitalizedLetterCharacterSet** - Titlecase Letters 범주의 문자를 포함하는 문자 집합을 반환합니다.
  - characterSetWithCharactersInString - 주어진 문자열의 문자를 포함하는 문자 세트를 반환합니다.
  - **characterSetWithRange** - 주어진 범위에서 유니코드 값을 가진 문자를 포함하는 문자 세트를 반환합니다.
  - **불법** 문자 집합 - 비문자 범주의 값을 포함하거나 유니코드 표준 버전 3.2에서 아직 정의되지 않은 문자 집합을 반환합니다.
  - **letterCharacterSet** - 문자 및 표시 범주의 문자를 포함하는 문자 집합을 반환합니다.
  - **lowercaseLetterCharacterSet** - 소문자 범주의 문자를 포함하는 문자 세트를 반환합니다.
  - **newlineCharacterSet** - 개행 문자를 포함하는 문자 세트를 반환합니다.
  - **punctuationCharacterSet** - 구두점 카테고리의 문자를 포함하는 문자 세트를 반환합니다.
  - **symbolCharacterSet** - 기호 범주의 문자를 포함하는 문자 세트를 반환합니다.
  - **uppercaseLetterCharacterSet** - 대문자 및 제목 대문자 범주의 문자를 포함하는 문자 세트를 반환합니다.
  - **whitespaceAndNewlineCharacterSet** - 유니코드 일반 범주 Z*, U000A ~ U000D 및 U0085를 포함하는 문자 집합을 반환합니다.
  - **whitespaceCharacterSet** - 인라인 공백 문자 공백(U+0020)과 탭(U+0009)만 포함하는 문자 세트를 반환합니다.

NSCharacterSet

Text and strings

```
#import <Foundation/Foundation.h>

int main() {
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   NSString *string = @"....Tutorials Point.com⋯..";
   NSLog(@"Initial String :%@", string);

   NSCharacterSet *characterset = [NSCharacterSet punctuationCharacterSet];
   string = [string stringByTrimmingCharactersInSet:characterset];
   NSLog(@"Final String :%@", string);

   [pool drain];
   return 0;
}
```

```
Initial String :....Tutorials Point.com.....
Final String :Tutorials Point.com
```

날짜와 시간

Dates and times

- NSDate 및 NSDateFormatter 클래스는 날짜 및 시간 기능을 제공합니다.
- NSDateFormatter는 NSDate를 NSString으로 또는 그 반대로 쉽게 변환할 수 있는 도우미 클래스입니다.
- 다음은 NSDate를 NSString으로 변환하고 다시 NSDate로 변환하는 간단한 예입니다.

날짜와 시간

Dates and times

```
#import <Foundation/Foundation.h>

int main() {
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSDate *date= [NSDate date];
   NSDateFormatter *dateFormatter = [[NSDateFormatter alloc]init];
   [dateFormatter setDateFormat:@“yyyy-MM-dd"];

   NSString *dateString = [dateFormatter stringFromDate:date];
   NSLog(@"Current date is %@“,dateString);
   NSDate *newDate = [dateFormatter dateFromString:dateString];
   NSLog(@"NewDate: %@",newDate);
  [pool drain];
   return 0;
}
```

```
Current date is 2013-09-29
NewDate: 2013-09-28 18:30:00 +0000
```

날짜와 시간

Dates and times

- 예제 프로그램에서 볼 수 있듯이 NSDate의 도움으로 현재 시간을 얻습니다.
- NSDateFormatter는 형식 변환을 처리하는 클래스입니다.
- 날짜 형식은 사용 가능한 데이터에 따라 변경될 수 있습니다.
- 예를 들어 위의 예에 시간을 추가하려는 경우 날짜 형식을 @"yyyy-MM-dd:hh:mm:ss"로 변경할 수 있습니다.

예외 처리

Exception handling

- 예외 처리는 기본 클래스 NSException이 있는 Objective-C에서 사용할 수 있습니다.
- 예외 처리는 다음 블록으로 구현됩니다.
  - **@try** - 이 블록은 일련의 명령문을 실행하려고 시도합니다.
  - **@catch** - 이 블록은 try 블록에서 예외를 잡으려고 시도합니다.
  - **@finally** - 이 블록에는 항상 실행되는 일련의 명령문이 포함되어 있습니다.

예외 처리

Exception handling

- 예외 처리는 기본 클래스 NSException이 있는 Objective-C에서 사용할 수 있습니다.
- 예외 처리는 다음 블록으로 구현됩니다.
  - **@try** - 이 블록은 일련의 명령문을 실행하려고 시도합니다.
  - **@catch** - 이 블록은 try 블록에서 예외를 잡으려고 시도합니다.
  - **@finally** - 이 블록에는 항상 실행되는 일련의 명령문이 포함되어 있습니다.

- 다음의 예제는 예외로 인해 프로그램이 종료되는 대신 예외 처리를 사용했기 때문에 계속 진행됩니다.

예외 처리

Exception handling

```
#import <Foundation/Foundation.h>

int main() {
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

  NSMutableArray *array = [[NSMutableArray alloc]init];

  @try  {
    NSString *string = [array objectAtIndex:10];
  } @catch (NSException *exception) {
    NSLog(@"%@ “,exception.name);
    NSLog(@"Reason: %@ “,exception.reason)
  }

 @finally  {
   NSLog(@"@@finaly Always Executes”);
  }

 [pool drain];
  return 0;
}
```

```
NSRangeException
Reason: *** -[__NSArrayM objectAtIndex:]: index 10 beyond bounds for empty array
@finally Always Executes
```

파일 처리

File handling

- NSFileManager 클래스의 도움으로 파일 처리를 사용할 수 있습니다. 이 예제는 온라인 컴파일러에서 작동하지 않습니다.

- 파일 처리에 사용되는 방법
  - **파일 액세스** 및 **조작** 에 사용되는 방법 목록 은 다음과 같습니다. 여기에서 원하는 작업을 수행하려면 FilePath1, FilePath2 및 FilePath 문자열을 필요한 전체 파일 경로로 바꿔야 합니다.

파일 처리

File handling

- 파일이 경로에 존재하는지 확인

```
NSFileManager *fileManager = [NSFileManager defaultManager];

//Get documents directory
NSArray *directoryPaths = NSSearchPathForDirectoriesInDomains
(NSDocumentDirectory, NSUserDomainMask, YES);
NSString *documentsDirectoryPath = [directoryPaths objectAtIndex:0];

if ([fileManager fileExistsAtPath:@""] == YES) {
   NSLog(@"File exists");
}
```

파일 처리

File handling

- 두 파일 내용 비교

```
if ([fileManager contentsEqualAtPath:@"FilePath1" andPath:@" FilePath2"]) {
   NSLog(@"Same content");
}
```

파일 처리

File handling

- 쓰기 가능, 읽기 가능, 실행 가능 여부 확인

```
if ([fileManager isWritableFileAtPath:@"FilePath"]) {
   NSLog(@"isWritable");
}

if ([fileManager isReadableFileAtPath:@"FilePath"]) {
   NSLog(@"isReadable");
}

if ( [fileManager isExecutableFileAtPath:@"FilePath"]) {
   NSLog(@"is Executable");
}
```

파일 처리

File handling

- 파일 이동

```
if([fileManager moveItemAtPath:@"FilePath1"    toPath:@"FilePath2" error:NULL]) {
    NSLog(@"Moved successfully”);
}
```

파일 처리

File handling

- 파일 복사

```
if ([fileManager copyItemAtPath:@“FilePath1"  toPath:@"FilePath2"  error:NULL]) {
    NSLog(@"Copied successfully");
}
```

파일 처리

File handling

- 파일 삭제

```
if ([fileManager removeItemAtPath:@"FilePath" error:NULL]) {
  NSLog(@"Removed successfully");
}
```

파일 처리

File handling

- 파일 읽기

```
NSData *data = [fileManager contentsAtPath:@"Path"];
```

파일 처리

File handling

- 파일 쓰기

```
[fileManager createFileAtPath:@"" contents:data attributes:nil];
```

URL 로딩 시스템

URL loading system

- URL 로딩은 URL, 즉 인터넷에서 항목에 액세스하는 데 유용합니다. 다음 클래스의 도움으로 제공됩니다.
  - NSMutableURL요청
  - NSURL연결
  - NSURL캐시
  - NSURL인증 챌린지
  - NSURL자격 증명
  - NSURLProtectionSpace
  - NSURL응답
  - NSURL다운로드
  - NSURL세션

- URL 로딩에 대한 예제는 코코아 애플리케이션 프로젝트에서 가능합니다. (생략)

빠른 열거

빠른 열거

Fast Enumeration

- 빠른 열거는 컬렉션을 열거하는 데 도움이 되는 Objective-C의 기능입니다.
- 따라서 빠른 열거에 대해 알기 위해서는 앞서 살펴본 Foundation Framework에서 제공하는 컬렉션에 대해 먼저 알아야 합니다.

컬렉션

Collections

- 컬렉션에는 여러 가지 유형이 있습니다. 그것들은 모두 다른 개체를 보유할 수 있다는 동일한 목적을 수행하지만, 대부분 개체를 검색하는 방식이 다릅니다.
- Objective-C에서 사용되는 가장 일반적인 컬렉션은 다음과 같습니다.
  - NSSet
  - NSArray
  - NSDictionary
  - NSMutableSet
  - NSMutableArray
  - NSMutableDictionary

빠른 열거 구문

Fast enumeration Syntax

```
for (classType variable in collectionObject ) {
  statements
}
```

빠른 열거 구문

Fast enumeration Syntax

```
#import <Foundation/Foundation.h>

int main() {
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSArray *array = [[NSArray alloc]
   initWithObjects:@"string1", @"string2",@"string3",nil];

   for(NSString *aString in array) {
      NSLog(@"Value: %@",aString);
   }

   [pool drain];
   return 0;
}
```

```
Value: string1
Value: string2
Value: string3
```

뒤로 빠른 열거

Fast Enumeration Backwards

```
for (classType variable in [collectionObject reverseObjectEnumerator] ) {
  statements
}
```

뒤로 빠른 열거

Fast Enumeration Backwards

```
#import <Foundation/Foundation.h>

int main() {
   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
   NSArray *array = [[NSArray alloc]
   initWithObjects:@"string1", @"string2",@"string3",nil];

   for(NSString *aString in [array reverseObjectEnumerator]) {
      NSLog(@"Value: %@",aString);
   }

   [pool drain];
   return 0;
}
```

```
Value: string3
Value: string2
Value: string1
```

메모리 관리

메모리 관리

Memory Management

- 메모리 관리는 모든 프로그래밍 언어에서 가장 중요한 프로세스 중 하나입니다.
- 객체의 메모리가 필요할 때 할당되고 더 이상 필요하지 않을 때 할당 해제되는 프로세스입니다.

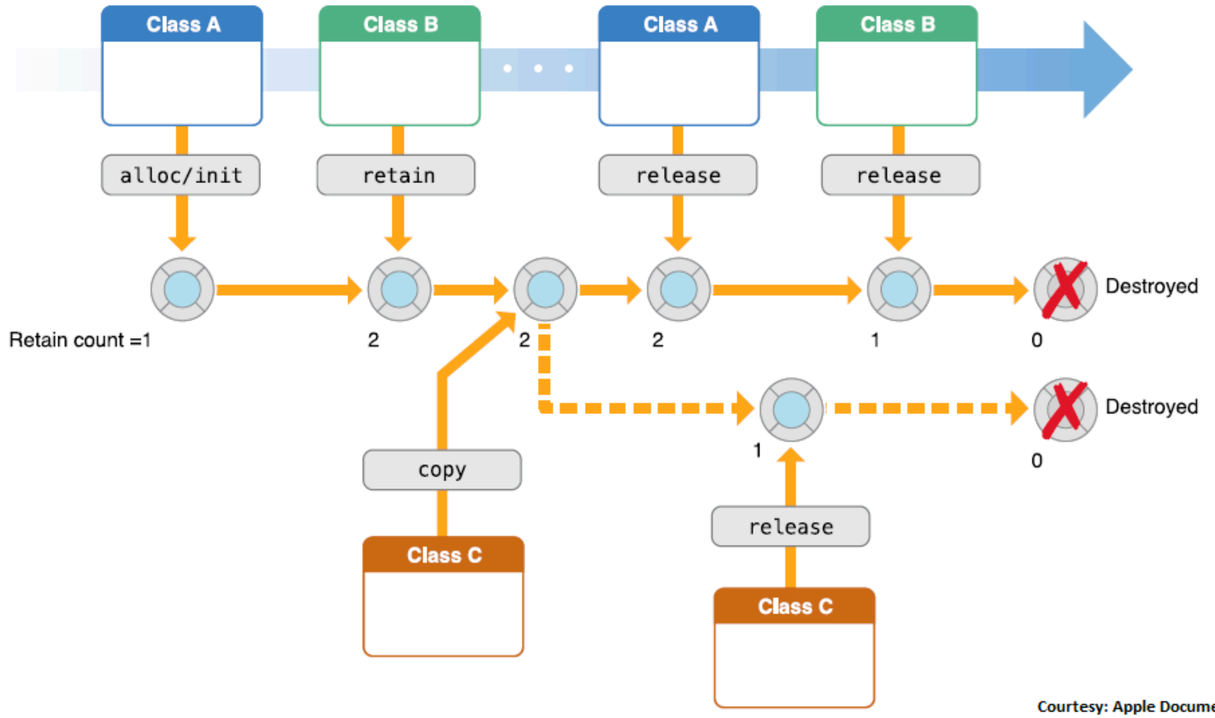
- 개체 메모리 관리는 성능 문제입니다.
- 응용 프로그램이 불필요한 개체를 해제하지 않으면 메모리 사용 공간이 늘어나고 성능이 저하됩니다.

- Objective-C 메모리 관리 기술은 크게 두 가지 유형으로 분류할 수 있습니다.
  - "수동 유지 해제" 또는 MRR
  - "자동 참조 카운팅" 또는 ARC

"수동 유지 해제" 또는 MRR

"Manual Retain-Release" or MRR

- MRR에서는 자체적으로 개체를 추적하여 메모리를 명시적으로 관리합니다.
- 이것은 Foundation 클래스 NSObject가 런타임 환경과 함께 제공하는 참조 카운팅으로 알려진 모델을 사용하여 구현됩니다.
- MRR과 ARC의 유일한 차이점은 전자에서는 유지 및 해제가 수동으로 처리되고 후자는 자동으로 처리된다는 것입니다.
- 다음 그림은 Objective-C에서 메모리 관리가 작동하는 방식의 예를 나타냅니다.

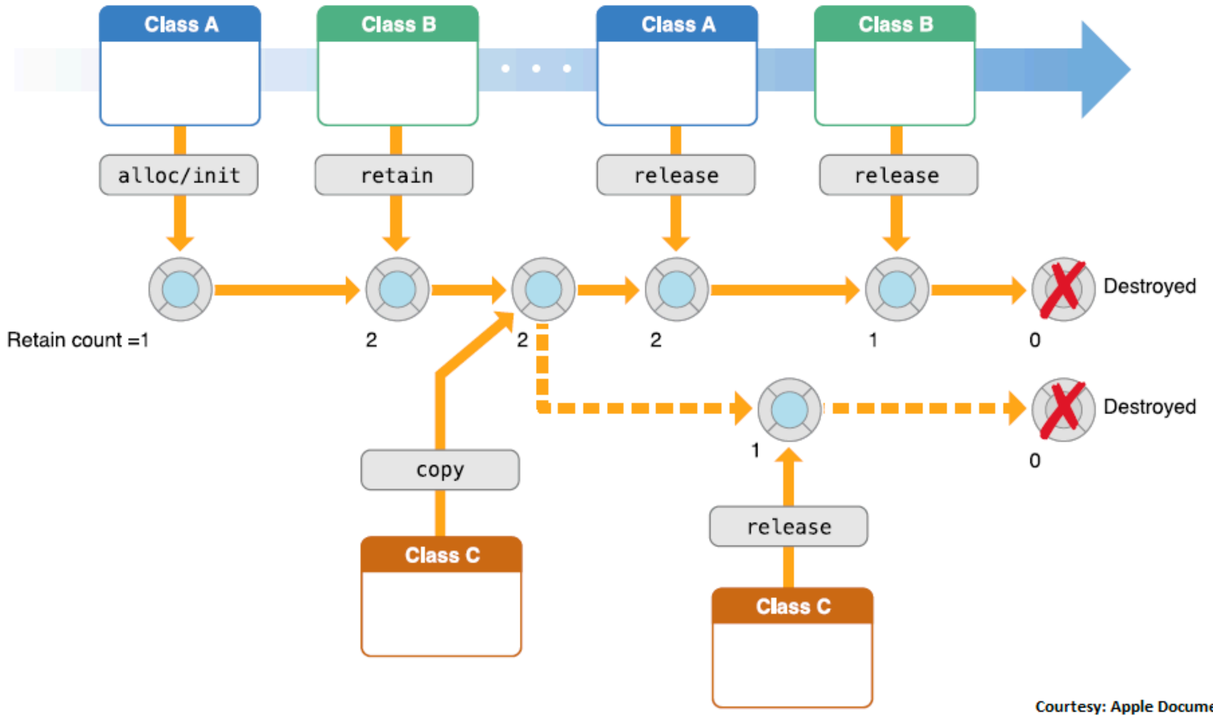


Courtesy: Apple Documentation

"수동 유지 해제" 또는 MRR

"Manual Retain-Release" or MRR

1. Class A 객체의 메모리 수명 주기는 위 그림과 같습니다. 보시다시피, 유지 횟수는 개체 아래에 표시되며 개체의 유지 횟수가 0이 되면 개체가 완전히 해제되고 다른 개체가 사용할 수 있도록 메모리가 할당 해제됩니다.
2. 클래스 A 객체는 먼저 NSObject에서 사용 가능한 alloc/init 메소드를 사용하여 생성됩니다. 이제 유지 횟수가 1이 됩니다.
3. 이제 클래스 B는 클래스 A의 객체를 유지하고 클래스 A의 객체의 보유 횟수는 2가 됩니다.
4. 그런 다음 클래스 C는 개체의 복사본을 만듭니다. 이제 인스턴스 변수에 대해 동일한 값을 사용하여 클래스 A의 다른 인스턴스로 생성됩니다. 여기서 유지 횟수는 1이며 원본 개체의 유지 횟수가 아닙니다. 이것은 그림에서 점선으로 표시됩니다.
5. 복사된 객체는 릴리스 메소드를 사용하여 클래스 C에 의해 해제되고 보유 횟수가 0이 되므로 객체가 파괴됩니다.
6. 초기 Class A Object의 경우 유지 횟수는 2이며 두 번 해제해야 소멸됩니다. 이것은 보유 횟수를 각각 1과 0으로 감소시키는 클래스 A와 클래스 B의 릴리스 문에 의해 수행됩니다. 마지막으로 개체가 파괴됩니다.



Courtesy: Apple Documentation

"수동 유지 해제" 또는 MRR

"Manual Retain-Release" or MRR

- 우리는 우리가 생성한 모든 객체를 소유합니다
  - 이름이 "alloc", "new", "copy" 또는 "mutableCopy"로 시작하는 메서드를 사용하여 객체를 생성합니다.

- 유지를 사용하여 객체의 소유권을 얻을 수 있습니다.
- 수신된 객체는 일반적으로 수신된 메서드 내에서 유효한 상태로 유지되며 해당 메서드는 객체를 호출자에게 안전하게 반환할 수도 있습니다.
- 우리는 두 가지 상황에서 유지를 사용합니다.
  - 접근자 메서드 또는 초기화 메서드의 구현에서 속성 값으로 저장하려는 개체의 소유권을 가져옵니다.
  - 다른 작업의 부작용으로 개체가 무효화되는 것을 방지합니다.

- 더 이상 필요하지 않을 때 우리는 우리가 소유한 객체의 소유권을 포기해야 합니다.
- 객체에 릴리스 메시지 또는 자동 릴리스 메시지를 전송하여 객체의 소유권을 포기합니다.
- 따라서 Cocoa 용어에서 객체의 소유권을 포기하는 것을 일반적으로 객체 "릴리스"라고 합니다.

- 소유하지 않은 개체의 소유권을 포기해서는 안 됩니다.
- 이는 명시적으로 명시된 이전 정책 규칙의 결과일 뿐입니다.

"수동 유지 해제" 또는 MRR

"Manual Retain-Release" or MRR

```objc
#import <Foundation/Foundation.h>

@interface SampleClass:NSObject
- (void)sampleMethod;
@end

@implementation SampleClass
- (void)sampleMethod {
   NSLog(@"Hello, World! \n");
}

- (void)dealloc  {
  NSLog(@"Object deallocated");
  [super dealloc];
}

@end
```

```objc
int main() {

   /* my first program in Objective-C */
   SampleClass *sampleClass = [[SampleClass alloc]init];
   [sampleClass sampleMethod];

   NSLog(@"Retain Count after initial allocation: %d",
   [sampleClass retainCount]);
   [sampleClass retain];

   NSLog(@"Retain Count after retain: %d", [sampleClass retainCount]);
   [sampleClass release];
   NSLog(@"Retain Count after release: %d", [sampleClass retainCount]);
   [sampleClass release];
   NSLog(@"SampleClass dealloc will be called before this");

   // Should set the object to nil
   sampleClass = nil;
   return 0;
}
```

```
Hello, World!
Retain Count after initial allocation: 1
Retain Count after retain: 2
Retain Count after release: 1
Object deallocated
SampleClass dealloc will be called before this
```

"자동 참조 카운팅" 또는 ARC

"Automatic Reference Counting" or ARC

- 자동 참조 카운팅 또는 ARC에서 시스템은 MRR과 동일한 참조 카운팅 시스템을 사용하지만 컴파일 타임에 적절한 메모리 관리 방법 호출을 삽입합니다.
- 우리는 새로운 프로젝트에 ARC를 사용할 것을 강력히 권장합니다.
- ARC를 사용하는 경우 일반적으로 이 문서에 설명된 기본 구현을 이해할 필요가 없지만 일부 상황에서는 도움이 될 수 있습니다.
- ARC에 대한 자세한 내용은 Apple의 ARC 릴리스 노트를 참조하세요.

- 위에서 언급했듯이 ARC에서는 컴파일러에서 처리하므로 릴리스 및 유지 메서드를 추가할 필요가 없습니다.
- 실제로 Objective-C의 기본 프로세스는 여전히 동일합니다.
- 내부적으로 유지 및 해제 작업을 사용하여 개발자가 이러한 작업에 대해 걱정하지 않고 더 쉽게 코딩할 수 있으므로 작성된 코드의 양과 메모리 누수 가능성이 모두 줄어듭니다.

"자동 참조 카운팅" 또는 ARC

"Automatic Reference Counting" or ARC

- Mac OS-X에서 MRR과 함께 사용하는 가비지 수집이라는 또 다른 원칙이 있었지만 OS-X Mountain Lion에서 폐지된 이후로 MRR의 대안이 되지 못하고 사라졌습니다.
- 또한 iOS 개체에는 가비지 수집 기능이 없었습니다.
- 그리고 ARC를 사용하면 OS-X에서도 가비지 수집을 사용하지 않습니다.

- 다음은 간단한 ARC 예제입니다.
- 이것은 ARC를 지원하지 않기 때문에 온라인 컴파일러에서는 작동하지 않습니다.

"자동 참조 카운팅" 또는 ARC

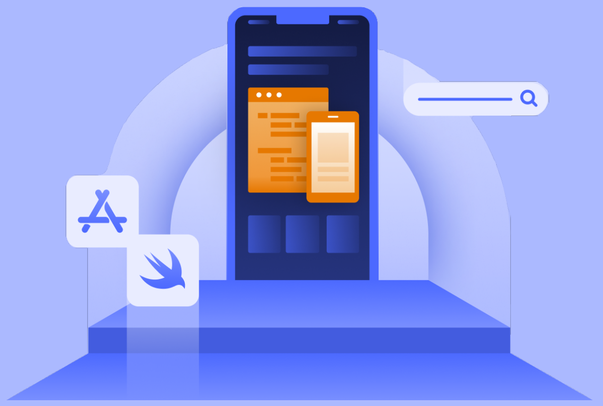
"Automatic Reference Counting" or ARC

```objc
#import <Foundation/Foundation.h>

@interface SampleClass:NSObject
- (void)sampleMethod;
@end

@implementation SampleClass
- (void)sampleMethod {
   NSLog(@"Hello, World! \n");
}

- (void)dealloc  {
  NSLog(@"Object deallocated");
}

@end
```

```objc
int main() {
   /* my first program in Objective-C */
   @autoreleasepool {
      SampleClass *sampleClass = [[SampleClass alloc]init];
      [sampleClass sampleMethod];
      sampleClass = nil;
   }
   return 0;
}
```

```
Hello, World!
Object deallocated
```

고맙습니다



LIKELION APP SCHOOL