Objective-C

Language Reference

박종암

jongampark@gmail.com

목차

들어가기 - 5 -

The Language - 7 -

Chapter 1. 객체(objects) - 9 -
- 1.1 object - 9 -
- 1.2 object pointer와 dynamic typing - 10 -

Chapter 2. Object Messaging - 11 -
- 2.1 메시지 호출법 - 11 -
- 2.2 Polymorphism - 13 -
- 2.3 Dynamic Binding 혹은 Late Binding - 13 -

Chapter 3. Classes - 15 -
- 3.1 상속 (Inheritance) - 15 -
  - 3.1.1 NSObject 클래스 - 16 -
  - 3.1.2 인스턴스 변수들을 상속하기 - 16 -
  - 3.1.3 method들을 상속하기 - 16 -
  - 3.1.4 Method overriding - 16 -
  - 3.1.5 Abstract class - 17 -
- 3.2 클래스 type - 17 -
  - 3.2.1 Static Typing (정적 타이핑) - 17 -
  - 3.2.2 Type Introspection - 18 -
- 3.3 클래스 객체 (Class Object) - 18 -
  - 3.3.1 인스턴스를 만들기 - 19 -
  - 3.3.2 Customization with Objective-C classes - 20 -
  - 3.3.3 변수와 클래스 객체(class object) - 21 -
  - 3.3.4 클래스 객체를 초기화하기 - 22 -
  - 3.3.5 루트 클래스의 메소드 - 23 -
- 3.4 소스 코드에서의 클래스 이름 - 24 -

Chapter 4. 클래스 정의하기 - 26 -
- 4.1 Interface - 26 -
  - 4.1.1 인터페이스 파일을 import하기 - 28 -
  - 4.1.2 다른 클래스를 언급하기 - 28 -
  - 4.1.3 Interface 파일의 용도 - 29 -
- 4.2 Implementation - 29 -
  - 4.2.1 인스턴스 변수를 억세스하기 - 30 -

4.2.2 인스턴스 변수의 scope .... - 31 -

Chapter 5. How Messaging Works .... - 34 -
5.1 Selector .... - 34 -
5.1.1. Messaging Error 처리 .... - 36 -
5.2 Hidden Argument .... - 37 -
5.3 Message to self and super .... - 37 -

Chapter 6. How to Extend Classes .... - 39 -
6.1 Category .... - 39 -
6.1.1 Category in Root Classes .... - 41 -
6.2 Protocol .... - 42 -
6.2.1 Protocol 이용 : 익명의 객체를 위한 Interface를 선언하기 .... - 44 -
6.2.2 Protocol 이용 : 비계층적 유사성(Non-Hierarchical Similarities) .... - 45 -
6.2.3 Protocol의 종류 .... - 46 -
6.2.3.1 Informal Protocol .... - 46 -
6.2.3.2 Formal Protocol .... - 48 -
6.2.4 Protocol Object .... - 49 -
6.2.5 프로토콜을 준수한다는 것 .... - 50 -
6.2.6 type checking .... - 51 -
6.2.7 protocol 안에서 또 protocol 쓰기 .... - 51 -
6.2.8 다른 protocol 을 언급하기 .... - 52 -

Chapter 7. Enabling Static Behaviors .... - 54 -
7.1 Static Typing .... - 54 -
7.1.1 Type Checking .... - 56 -
7.1.2 Return과 Argument 타입 .... - 56 -
7.1.3 상속된 클래스에 대한 Static Typing .... - 57 -
7.2 Method의 주소를 얻기 .... - 57 -
7.3 객체의 data structure를 얻기 .... - 58 -

Chapter 8. Exception Handling과 Thread Synchronization .... - 60 -
8.1 Exception 처리하기 .... - 60 -
8.1.1 Exception 던지기 .... - 61 -
8.1.2 Exception 처리하기 .... - 61 -
8.2 Thread 실행을 synchronizing하기 .... - 62 -

Chapter 9. Objective-C와 C++을 같이 사용하기 .... - 64 -
9.1 Objective-C와 C++의 언어 기능을 함께 사용하기 .... - 64 -
9.2 C++의 keyword때문에 생기는 모호성과 충돌 (C++ Lexical Ambiguities and Conflicts).. - 67 -

Runtime System .... - 69 -

Chapter 10. 메모리 관리 .... - 71 -
10.1 객체를 할당하고 초기화하기 .... - 71 -
10.1.1 반환된 객체 .... - 72 -
10.1.2 init…의 인자(argument) .... - 73 -
10.1.3 Class를 coordination하기 .... - 73 -
10.1.4 Designated Initializer (최종적으로 호출될 initializer) .... - 75 -
10.1.5 할당과 초기화를 같이 하기 .... - 76 -
10.2 객체의 소유 관계 .... - 77 -
10.3 객체를 계속 가지고 있기 (Retaining Object) .... - 78 -
10.3.1 뺑뺑이 reference (Cyclical References) 다루기 .... - 79 -
10.4 객체를 없애기(Deallocation) .... - 79 -
10.4.1 공유된 객체를 release하기 .... - 79 -
10.4.2 인스턴스 변수들을 release하기 .... - 80 -
10.4.3 나중에 release하도록 객체에 표시해 놓기 .... - 81 -

Chapter 11. 메시지 forwarding .... - 82 -
11.1 전달(Forwarding)과 다중 상속(Multiple Inheritance) .... - 83 -
11.2 대리 객체(Surrogate Object) .... - 84 -
11.3 전달과 상속 .... - 84 -

Chapter 12. 동적 로딩(Dynamic Loading) .... - 86 -

Chapter 13. 원격 메시징(Remote Messaging) .... - 87 -
13.1 분산 객체 (Distributed Objects) .... - 87 -
13.2 언어에서 받는 지원 .... - 88 -
13.2.1 동기 메시지와 비동기 메시지(Synchronous and Asynchronous Messages).. - 89 -
13.2.2 포인터 인자(Pointer Arguments) .... - 90 -
13.2.3 프록시와 카피본 (Proxy and Copies) .... - 92 -

Chapter 14. Type Encoding .... - 94 -

맺는말 .... - 97 -

들어가기

이 문서의 목적

이 문서는 개인적인 관심에서 쓰여지기 시작했습니다. Apple 은 Mac OS X 과 Objective-C 그리고 Cocoa 라는 매력적인 개발 언어 환경을 가지고 있지만 한국 개발자들을 위한 문서는 만들지 않는다는 점이 답답했습니다. 이게 왜 문제가 되는지 생각해 보면, 자명합니다. 물론 영어를 잘 아는 사람들이 개발을 하게 되면, 개발 자체는 하게 됩니다. 하지만 원어민이 아닌 이상, 아무래도 문서를 보는 속도는 떨어지게 되어 있습니다. 예를들어 Windows 개발자가 Mac 에 관심이 있어, Apple 사의 웹 사이트에 들어가서 개발 문서를 한번 본다고 합시다. 흥미를 느껴서 좀 보게 될것일겁니다. 만약 진득하니 계속 볼수있는 사람이라면 모르겠지만 필자같이 부담스러운것에 지치는 사람들은, 그리고 하던 일이 맥이 끊기면 다음에 이어서 하기 힘든 사람들은 어느 정도 보다가 그칠것입니다.
하지만 만약 한글로 되어 있다면 순식간에 후다닥 보게 되므로, 아직 열의가 남아있을때, 한번 시험삼아 짧은 코딩이라도 해보게 될것입니다. 그 정도까지가면 그 다음엔 잠시 덮어 놓더라도 기억이 남아 있게 될테고, 짜투리 시간에 조금씩 더 해 볼수가 있을 겁니다. 그렇게 되면 Mac 개발에 있어서 어느 정도의 저변이 확대가 될 가능성이 조금이라도 더 있지 않을까 싶습니다.

모쪼록 관심을 가져주시는 osxdev.org 에 계신 회원분들에게 감사를 표합니다.

이 문서의 내용

이 글은 약 3 개월 이상의 시간을 들여서 쓴 것입니다. 맨처음엔 단순한 노트에서 시작을 했고, 중간에는 C++과 비교를 해서 C++프로그래머들의 학습 속도를 빠르게 할수 있도록 했습니다만, 시간이 많이 걸려, 전체적으로 그렇게 하지는 못했습니다. 만약 그런 책이 나온다면 한쪽에 익숙해지면 다른 한쪽을 잊는 그런 것을 방지하기에도 좋지 않을까합니다. 이건 예전에 Java 프로그래밍을 많이 했을때 있었던 경험인데, 비슷하게 생긴 언어가 혼동도 되기 쉬운만큼, 언젠가 이런 책을 써 보고 싶습니다.

쓰는 시간이 길어지다보니 말투도 일관되지 않고 포맷도 일관되지 않게 되어 다시 처음부터 포맷을 맞추어 주어야 하는 작업을 하게 되었습니다. 말투 바꾸는 것은 너무 많아서 적당히만 하기로 했습니다.

원래는 빨리 읽고 익숙해지는 것을 목표로 하였으나, Category 나 Protocol 과 같은 설명에서는 Apple 의 설명을 그대로 번역하는 정도가 되었습니다. 물론 가감이 없지는 않으나 거의 같다고 보시면 되겠습니다. 그러다보니 문서가 꽤 길어지게 되었습니다. 그리고 좀더 프로그래밍 예가 있었으면 싶지만, 그것도 그냥 Apple 의 예를 가지고 왔습니다. 일반 C/C++책처럼 아는 부분이라도 하나 하나 해보게 만들면 자연스레 Objective-C 에 익숙해질텐데 하는 아쉬움이 남습니다.

내용의 순서는 Apple 의 문서에서 보이는 순서와 같습니다. 아무래도 체계가 잡혀있는 문서여서 따라하다 보니 그리되었습니다. 읽어 보시면 아시겠지만, 최종적으로는 거의 Apple 문서의 번역본이 되었습니다. 하지만 처음 Object Oriented Programming 에 대한 것은 누락을 했습니다. 아마도 Apple 의 Objective-C 에 관심을 두실 정도의 분들이라면 이미 C++에서의 객체 지향형 프로그래밍에 대해서 아실 것이라 생각되어, 사실 크게 다른 것이 없으므로 누락을 했습니다. 제가 보는 관점에서의 차이점에 대해서는 간략히 서술을 하였습니다.

저는 전문적인 Objective-C 프로그래머가 아니기에 문서의 내용이 많이 부족할 것이고, 설명해 놓은 관점도 많이 부족할 것입니다. 하지만 빨리 읽을 수 있는, 그리고 C++이나 C 에서 빨리 적응할 수 있는 용도로 쓰일 수 있다면 그것으로 만족하겠습니다.

이 Objective-C Language Reference 로 조금이나마 Mac 플랫폼에 대한 개발자 저변 확대, 그리고 C++과는 조금 다른 객체 지향 방식에 접근을 해 보실 수있는 기회가 제공되기를 바랍니다.

The Language

Objective-C 언어는 C++과 마찬가지로 Object Oriented Programming 언어입니다. 그 모태는 많은 분들이 아시다시피 SmallTalk 입니다. 하지만 이 언어는 C++과 다른 점도 있습니다.

- 기존 C 에 최소한의 확장으로 OOP 기능을 넣었다
- 굉장히 dynamic 한 언어이다.
- MVC (Model-View-Controller) 모델을 주로 한다
- Message send/receive 이 기본 패러다임의 주요 요소이다
- 클래스 디자인시 Get / Put 개념이 많이 쓰인다
- 상속과 같은 수직적 클래스 확장 외에도 카테고리와 같은 개념을 이용하여 수평적 확장도 가능하다.
- 확장의 대상이 되는 클래스를 위한 소스코드가 없어도 확장 가능하다.
- 진정한 모듈러 프로그래밍이 가능하다.
- Garbage collection 을 자체에서 지원한다.

이 특징은 SmallTalk 와 맥락을 같이 한다. 즉 대부분의 개념이 이미 SmallTalk 에 있는 것을 그대로 받아온 것이다. 물론 이제는 C++도 RTTI 같은 것을 지원하므로 runtime 시에 introspection 같은 것이 가능하며 상당히 dynamic 한 측면을 보인다. 그리고 ActiveX 프로그래밍을 할때도 get/put 패러다임을 쓴다. 그러므로 이제는 어떤 면에선 Objective-C 가 특별하게 보이지는 않을 수도 있다. 하지만 이 Objective-C 가 벌써 한 10 년 이전에 이런 모습을 갖추었다는 것을 고려한다면 참 대단한 일이 아닐 수 없다. 또한 C++ 프로그래머에게 있어서 놀라운 점은, 이 언어는 기존 C 에 많은 것을 첨가하지 않았는데도 OOP 를 훌륭하게 소화해 낸다는 것이다.

하지만 아마도 제일 놀라운 점은, 바로 진정한 모듈러 프로그래밍이 가능하다는 점일 것이다. 즉 정의되지 않은 클래스에 대한 프로그래밍이 가능하며, 기존 클래스를 확장하려할 때, 그 클래스에 대한 소스코드가 꼭 필요하지는 않다. 또한 respondsTo:와 같은 메시지를 사용함으로써 runtime 시에 대상이 되는 클래스가 해당 메시지를 처리하는지 등을 검사할 수 있으므로 보다 더 유연한 상황대처가 가능하다. 이런 메커니즘으로 프로그램을 모듈화해서 꼭 레고 부품 만들 듯이 만들어 넣을 수가 있다는 것이다. 이것은 C++에서 이야기하는 Encapsulation 의 진정한 모습이라고 할 수 있다. C++에서의 약점은 바로 Data Hiding / Data encapsulation 이 실제적으로 그다지 잘 보장되지 않는다는 점이었다. 그리고 엔지니어링 관점에서 소프트웨어 개발 속도의 향상과 유지 보수를 그다지 잘 높이지 못했다는 점이 약점으로 대두되었었다.

또한 method invocation 이 아닌 message passing 이라는.. 약간의 표현차이지만 그 표현차이로 자극하는 뭔가가 보다 더 OOP 의 실제 모습에 가깝다는 면도 이 언어의 강한 측면이라고 볼 수있겠다.

하지만 안타깝게도 이런 훌륭한 OOP 언어인 Objective-C 는 시장에서 그다지 환영받지 못했다. 물론 ISO 의 표준화 노력도 있었으나 C++만큼 널리 받아들여지지 않았다. 여기에는 현실적 실용적 문제가 있었을 것이라 보인다. 동적인 언어인 만큼 late binding 이 많아지게 되므로 유연은 해지지만

PC 의 속도가 느렸던 시절에, 최대한으로 CPU 파워를 뽑아 내려 하고 싶었던 것과 반하는 면도 있었고, 사람들이 이렇게까지는 필요로하지 않았었다는 면, 그리고 강하게 밀던 NeXT 의 사업 실적이 신통치 않았다는 것도 이유가 될 수 있을지 모르겠다.
이상하게도 SmallTalk 에 기반을 둔 언어들은 연구 측면에선 강한 세력을 보이고 있으나 실용면에선 좀 전통적으로 약세라는 면도 이상한 기운이라면 기운이겠다.
요새는 역시 SmallTalk 에서 강한 영향을 받은 Ruby 가 심심치않게 인기를 얻고 있으니, 그런 맥락에서 Objecive-C 를 배워보는 것도 좋겠다.

어쩌면 MacOS X 을 제외하고는 실질적으로 쓰지 않는 Objective-C 를 왜 배우느냐고 생각하실 분들이 있을지도 모르겠다. 아니 필자도 그런 생각을 종종한다. 하지만 그래도 가치가 있는 언어라고 생각하며, 요새 C++도 Objective-C 의 많은 면을 채택하고 있으며, MS 의 개발 패러다임도 많이 Get/Put, MVC 모델에 입각하는 측면이 있으므로, 이 Objective-C 를 배우면 그 OOP 의 정수를 한번 맞보지 않을까 생각해본다.

Chapter 1. 객체(objects)

Objective-C 는 C 와 주로 객체에 대한 부분만이 다릅니다. 물론 method 를 호출하는 문법은 정말 다른데, 이 둘을 제외하고는 C 와 동일합니다. 그것은 Objective-C 가 C++처럼 새로운 언어라기보다는 C 에 대한 superset 이기 때문입니다.

그러므로 이 문서에선 Objective-C 만의 기능에 대해서 알아보기로 합시다.

1.1 object

OOP 는 객체를 중심으로 프로그래밍을 합니다. 그러므로 객체에 대한 용어를 우선 알고 가는 것이 좋습니다.
C++에서 클래스 내의 변수는 member variable 이라고도 말하는데, Objective-C 에서는 *instance variable* 이라고 합니다. 때때로 이것은 OOP 적 표현으로 property 라고도 한다는 점을 생각해둡시다.
그리고 C++에서 클래스의 멤버 함수는 Objective-C 에서는 *method* 라고 합니다. C++의 용어와 Objective-C 에서의 용어는 OOP 관련 책자에서 자주 인용되므로 다 알아두는 것이 좋을 것입니다.

보통 instance variable 은 해당 객체의 access 메소드를 이용해서 접근합니다. 즉 get 함수와 put 함수를 사용하게 되는데, 보통 get 함수는 해당 instance variable 과 같은 이름으로 하기도 합니다.
즉 다음의 예를 봅시다.

```
@interface Song : NSObject
{
        NSString *name; // instance variable
        NSString *artist;

}
- (NSString *) name; // instance method
- (NSString *) artist;
```

이것을 사용해서 Song 클래스에 있는 name 이 어떤 값을 가지고 있는지 알고 싶다면 다음과 같이 합니다.

```
Song mySong;
…
[mySong name];
```

이렇게 Objective-C 에서는 instance variable 과 같은 이름의 method 를 가지는 것이 가능합니다. C/C++식으로 말하자면 instance method 인 (NSString *) name 은 char *name(void) 와 같은 식인 것입니다. 맨처음엔 인자를 괄호안에 쓰지 않는 Objective-C 스타일이 좀 이상하게 보일 수도 있지만, 인자가 길어지고 많아질때는 가독성면에서 상당히 도움이 된다는 것을 알 수가 있을겁니다.

아무튼 여기에선 이런 식으로 객체의 instance variable 을 억세스한다는 것을 알아두는 것으로 넘어갑시다. 물론 꼭 instance variable 과 같은 이름으로 access method 의 이름을 정하지 않아도 됩니다. 어떻게 정하라라는 규칙은 없습니다. 여러분이 더 좋다고 생각하는 명명법이 있다면 그렇게 하십시오. 하지만 많은 Objective-C 책들은 이런 식으로 하는 것으로 보아, 은근히 이런 코딩 스타일을 유도하는 것 같습니다.

1.2 object pointer 와 dynamic typing

Objective-C 는 동적인 언어입니다. 이 말은 runtime 시에 객체가 변할 수 있다는 것을 의미하며, 또한 해당 객체에 대한 method 도 변할 수 있습니다.. 이것은 Apple 에서 만들던 Dylan 의 설명서를 보면 쉽게 이해할 수 있을것입니다. 아무튼 이렇게 동적이다보니, 기존의 static typing 을 하던 방식으로 객체에 대한 포인터를 사용하면 불편한 면이 있을 수 있겠습니다. 그래서인지 Objective-C 에는 모든 객체를 두루 포인팅할 수 있는 타입을 새로 정의해 놓았는데 바로 id 입니다. 다음의 예를 봅시다.

```
id mySong = [[Song alloc] init];
```

이렇게 하면 mySong 은 Song 이라는 타입의 객체를 가르키게 됩니다.
물론 static typing 을 써서 다음과 같이 할 수도 있습니다.

```
Song mySong = …
```

이것은 흡사 C/C++의 void *와 비슷합니다.
만약 아무것도 안가르키고 있다면 nil 이라는 이미 정의된 keyword 를 null 객체를 가르키기 위해서 사용할 수 있습니다.

```
id mySong = nil;
```

id 타입은 객체에 대해서 어떤 정보도 가지고 있지 않습니다. 단지 가르키고 있는게 객체라는 것만을 의미할 뿐입니다. 그러므로 id 로 선언된 변수는 어떤 객체라도 가르킬 수 있습니다.
이것은 C/C++ 관점에서 보자면 문제를 일으킬 수 있습니다. 즉 가르키고 있는 객체가 어떤 형태인줄 알아야 뭔가를 할 수 있을텐데, 이것만으로는 알아낼 방도가 없기 때문입니다.
그래서 *isa* 라는 instance variable 을 묵시적으로 가지고 있게 됩니다. 이것은 실제 그 객체가 어떤 형인지를 기록해 놓는 것인데, 이것을 이용해서 runtime 시에 알아낼 수 있게 됩니다.
이런 메커니즘을 이용해서 dynamic typing 이 가능해 집니다.
이 변수는 Xcode 에서 디버깅을 해보면 볼 수가 있습니다.

Chapter 2. Object Messaging

Object messaging 은 쉽게 말하자면 한 객체의 멤버 함수인 method 를 호출하는 것입니다. 보통 Objective-C 에서는 해당 객체에 메시지를 보낸다라고 표현을 합니다. C++에서는 멤버함수를 호출한다라는 표현을 쓰고 있습니다. 바로 이 표현이 Objective-C 의 OOP 스타일과 C++의 OOP 스타일의 주요 차이점이 되지 않나 싶습니다. 물론 이외에도 몇몇 다른점이 있지만 OOP 구성을 하는 전체적인 그림의 차이는 바로 이 표현에서 보입니다. Apple 의 Objective-C 문서를 보면, 그리고 SmallTalk 이나 기타 Objective-C 문서를 보면, 객체를 원으로 표현하고, 그 객체에 메시지를 보내는 것과 같은, 객체의 네트워크로 프로그램을 표현하는 것을 많이 보실 수 있을 겁니다. 바로 이런 표현을 도식화하면 그렇게 됩니다.

이런 것을 생각하면서 접근을 하게 되면 Objective-C 스타일의 OOP 에 좀더 빨리 익숙해질 수 있을 것이라고 생각합니다.

2.1 메시지 호출법

한 객체에 메시지를 보내는 법, 다시 말하자면 그 객체의 method 를 호출하는 법은 다음과 같습니다.

[객체 method]

즉 이것은 C++스타일의 *class.member_function* 혹은 *class->member_function* 과 같습니다.
하지만 이것을 좀더 Objective-C 다운 표현으로 바꿔 보자면 다음과 같습니다.

[receiver message]

즉 호출대상이 되는 객체는 보내는 메시지를 받는 것이 되므로 receiver 라 하고, 그 객체의 method 를 메시지로 보내기 때문에 호출되는 메소드에 대해서는 message 라고 표현을 합니다. 생각의 차이지만 이렇게 표현이 달라진다는 점이 재미납니다. 이제부터는 Objective-C 스타일로 모든 표현을 해 나가겠습니다.

여기서 또하나 재미난 점은 메시지의 receiver 의 앞과 message 의 뒤에 [,]의 괄호로 둘러 쌓아 놓은 것입니다. 이런 스타일은 참 낯설기 그지 없어서, 많은 사람들이 Objective-C 가 이상하다라고 할때 종종 언급하는 것입니다. 물론 처음에 볼때는 참 낯설어 보입니다. 하지만 하지만 익숙해지면 나름대로 깔끔해 보이고, 아 객체에 대해서 뭔가를 하는구나가 눈에 확 띕니다.

물론 부정적인 면도 없지는 않습니다. C++스타일로 포인터냐 아니냐에 따라 ->나 .를 쓰게 되면, 좌에서 우로 쓰면서 계속 연달아 쓸 수 있습니다. 쓰면서.. 아 이건 객체지 하면서 쓸 수 있단 뜻입니다.하지만 Objective-C 는 receiver 를 쓰기 전에, 아 객체에 메시지를 보내야지란 생각을 먼저해야 하고 그 다음에 [을 쓰고서 시작해야합니다. 그러지 않으면 일단 뒤에까지 죽 썼는데, 아차! 하면서 앞으로 다시 커서를 옮기고 괄호를 써야겠죠. 이 표현은 nesting 될 수 있는데 그 경우엔 더 합니다.

즉 다음을 봅시다.

[[[A child] child] child]

여기서 child 가 객체들이 내부적으로 가지고 있는 어떤 객체를 반환한다고 합시다. 이럴 경우엔 A 앞에 괄호가 세개나 들어가므로 좀 writability 가 떨어지기는 합니다.

맨처음 Objective-C 를 배울때, 필자같은 경우, 이런 표현때문에 좀 고생을 했고, 여전히 C/C++ 프로그래밍을 하는 입장이다보니 좀 불편하기까지 합니다만, 그래도 Objective-C 를 쓸때는 어쩔 수 없습니다. 개인적으로 Xcode 의 editor 가 이런 것을 자동으로 괄호를 쳐 주었으면 편하겠다 싶습니다.

이제 method 가 인자를 가질때를 살펴봅시다.

```
[myRect setWidth:10.0 height:15.0]
```

즉 myRect 라는 어떤 객체에 그 사각형의 폭과 높이를 10.0 과 15.0 으로 세팅을 하는 것입니다. 이런 표현도 낯섭니다. setWidth 까지는 함수의 이름이겠거니하고 별 부담이 없이 받아들여지는데 height 부분은 낯섭니다. 이게 함수 이름의 부분인지 아니면 뭔지..
Objective-C 는 인자에 이름을 붙일 수 있습니다. 위의 method 를 정의 한다면 이렇게 될 것입니다.

```
- (void) setWidth: (float ) width height: (float) height
```

주의깊게 이 proto type 을 살펴 봅시다. 제일 앞의 –는 이 method 가 instance method 라는 것을 의미합니다. 즉 C++의 멤버함수와 같습니다. 그리고는 이 method 의 반환값에 대한 형태가 괄호를 이용해서 (void)라고 되어 있습니다. 그 뒤에는 method 의 이름인 setWidth 가 나옵니다. 이제부터는 이 method 의 인자를 쓰게 되는데, 우선 :를 써서 인자를 구별합니다.
그리고 각 인자의 형은 괄호를 이용해서 표현을 합니다. 즉 (float) width 는 이 메소드의 첫번째 인자는 width 라는 변수를 통해서 전달되며 그 형은 float 입니다. 그리고 두번째 인자를 쓰는데, 여기서 특이한 점이 인자에 이름을 붙인다는 것입니다. 즉 height: 부분이 두번째 인자의 이름입니다. 그리고 그 다음에 두번째 인자를 씁니다.
불편하십니까? 무척 이상하게 보일겁니다. C/C++에 익숙해져 있다면, 어떤게 인자이고 어떤게 이름인지 눈에 잘 안띌 겁니다. 왜냐하면 C/C++은 괄호안에 각 인자를 컴마를 이용해서 넣게 되어 있기 때문입니다. 즉 괄호 안의 것들은 죄다 인자인데, Objective-C 는 어디서부터가 인자인지 눈에 잘 안들어옵니다.
하지만 이것은 익숙해지기 나름입니다. 이 Objective-C 스타일에 익숙해지시면, 나름대로 깔끔해 보입니다. 이런 표현은 나름대로 장점을 가집니다.

또한 한가지 이상한 점이 있습니다. 바로 method 를 정의하는 앞에 –가 붙어 있는 것입니다. 이것은 다음과 같습니다.

-	Instance method
+	Class method

즉 +를 붙이면 C++의 클래스에서 static 으로 선언한 것과 같은 효과를 볼 수가 있습니다.

2.2 Polymorphism

Objective-C 도 OOP 언어인만큼 Polymorphism 과 뗄래야 뗄 수 없는 관계에 있습니다. 하지만 지원하는 polymorphism 은 C++의 그것과는 사뭇 다릅니다. 일단 이곳에서는 메시지 전달에 대한 것을 다루는 만큼 polymorphism 도 C++과는 다르게 "메시지 전달"의 관점에서 살펴봅시다. 그리고 나서 일반적인 polymorphism 에 대해서 알아봅시다.

C++과 마찬가지로 Objective-C 에서는 일단 객체가 다르다면 그들 사이에 같은 메소드를 가질 수 있습니다. 물론 인자가 완전히 동일할 수도 있고 다를 수도 있습니다. 이런 형태도 Objective-C 에서는 polymorphism 에 해당합니다. 왜 그런가하면 "메시지 전달"이라는 점을 생각해 보면 알 수 있습니다. 이를테면 여러분의 코드가 display 란 메시지를 Rectangle 이란 객체와 Cube 라는 객체에 보낸다고 합시다. 비록 두 객체가 다 display 란 메소드를 가지고 있다고 해도, 하는 일은 아마도 다를겁니다. 하지만 보내는 측에서는 id 로 두 객체를 가르킬수가 있으며, 아무런 문제없이 display 란 메시지를 다 보낼 수 있습니다. 즉 다시 말하자면 다른 객체가 같은 메시지에 대해서 반응하므로 이것도 polymorphism 에 해당합니다.

보통 C++에서 polymorphism 이라고 하면 더 엄밀하게는 parametric polymorphism 을 이야기 합니다. 즉 멤버 함수 이름은 같지만 그 인자는 다른 그런 경우입니다. 이런 것은 Objective-C 도 지원을 합니다. 또한 Objective-C 는 C++처럼 parent 클래스 혹은 super class 의 method 도 overriding 할 수 있습니다.

하지만 Objective-C 는 연산자 오버로딩 (operator overloading)은 지원하지 못합니다. 이것은 ad-hoc polymorphism 이라고도 하는데, 이것을 잘 이용하면 덧셈 부호를 행렬의 덧셈에 사용한다던가 하는 식으로 상당히 깔끔하게 보이고 편한 코딩을 할 수가 있습니다. 아쉽게도 Objective-C 는 연산자 오버로딩은 지원하지 못합니다.

2.3 Dynamic Binding 혹은 Late Binding

이렇게 Objective-C 에서 어떤 객체에 메시지를 보낼 수 있게 되는데, 객체와 해당 메시지는 언제 엮여지게 될까요? 그것은 runtime 시에 엮어집니다. 보통 Objective-C 프로그래밍을 하면 객체는 id 의 형태로 받고 그것에 대해서 메시지를 보내기 때문에, compile 시에는 그 객체의 형태가 구체적으로 뭐가 되는지 알 수 없습니다. 그러므로 runtime 시에 하게 됩니다.
물론 이런 동적 바인딩 (dynamic binding)은 장점도 있고 단점도 있습니다. 장점으로는 대단히 유연하다는 것입니다. 예를 들자면 여러가지 형태를 담는 리스트나 큐를 만들기가 쉽습니다. 하지만 단점도 있습니다. 우선 컴파일 시에 에러 메시지가 뜨지 않습니다. 그러므로 잘못 만들면 런타임 시에

찾기 힘든 에러가 발생할 수도 있습니다. 그 다음 문제로는 성능의 문제가 있겠습니다. 적어도 static binding 을 하게 되면 컴파일시에 에지간한 잔작업은 미리 해 놓기 때문에 런타임시에는 말그래도 주로 수행하는데만 CPU 타임을 쓰게 되지만, 동적 바인딩을 하면 그렇지 않습니다.

물론 Objective-C 는 여러분의 선택에 따라 동적 바인딩과 정적 바인딩을 할 수있게 해줍니다. 이것은 뒤의 Chapter 7. Enabling Static Behaviour 를 참고 하시기 바랍니다.

Chapter 3. Classes

앞에서는 객체에 대한 일반적인 설명과 객체의 method 들에 대해서 알아보았습니다. 순서가 뒤바뀐 면이 있군요. 이 장에서는 그런 객체를 어떻게 만들것인지, method 는 어떻게 선언되는지에 대해서 알아보도록 하겠습니다. 즉 객체의 선언과 구현에 대해서 알아보는 것이 되겠습니다.

Objective-C 에서 객체를 정의하려면, 그 클래스를 정의하면 됩니다. 즉 C++에서 클래스를 정의하는 것과 개념적으로 완전히 동일합니다. 컴파일러는 실제로 각 클래스에 대해서 클래스 객체(class object)라는 것을 하나만 만듭니다. 이 클래스 객체는, 그 타입으로 만들어질 객체들을 어떻게 만들어야 할지에 대한 정보를 다 가지고 있으므로, factory object 라고도 보통 불리곤합니다. 클래스 객체는 다시 말하자면 해당 클래스의 컴파일된 버젼입니다. 그 클래스 객체가 만드는 객체는 해당 클래스의 instance 라고 합니다. 실제 돌아가는 프로그램에서 주된 역할을 하게 되는 객체들은 사실 이렇게 런타임시에 클래스 객체를 이용해서 만들어지는 instance 들입니다.

NOTE : 이 부분은 Apple 의 문서를 바로 번역한 것입니다. 이처럼 클래스와 객체(object), 그리고 instance 를 쉽고도 명확하게 설명한 문서는 그리 많지 않은거 같습니다. C++에서는 항상 이 개념들을 명확하게 설명하지 않는 경향이있죠. 사실 그래도 별 문제는 없겠습니다. 결국 추상적인 클래스를 논리적으로 보느냐 런타임시의 실질적인 사물로 보느냐에 따른 용어이기 때문입니다. 컴파일러 제작자라면 이 개념을 명확히 구분할 필요가 있겠지만, 일반적으로는 그냥 섞어 사용해도 대개 의미 전달에는 크게 왜곡이 없으리라 봅니다. 물론 AT&T 의 C++책에는 이 개념이 설명이 나와 있지만, 이렇게 쉽게 이해되게는 나와있지 않았었습니다. 이 점이 여러가지를 배워둘때 얻을 수 있는 장점이랄 수 있겠습니다.

클래스의 각 instance 는 같은 method 를 갖습니다. 즉 같은 주물(鑄物)에서 만들어진 같은 instance 변수들을 가지게 됩니다. 이때 각 객체들은 그 instance 변수들을 자체적으로 내부에 가지게 되지만, method 들은 서로 공유하게 됩니다.

클래스에 대한 코딩 스타일은, 전통적으로 그 이름을 대문자로 시작합니다. 예를 들자면 Rectangle 처럼 말입니다. 그리고 그 instance 의 이름은 보통 소문자로 시작합니다. 이럴 경우에 myRect 가 될 수 있겠습니다.

3.1 상속 (Inheritance)

OOP 언어인만큼 Objective-C 도 상속을 지원합니다. 즉 한 클래스를 만들때, 다른 클래스와 유사한 부분이 있다거나 같은 부류인데 좀 작은 개념이라고 생각될때, 다른 클래스의 instance 변수와 method 를 그대로 받아서 새로운 클래스를 만들 수 있는 것입니다. 물론 이때, 다른 클래스로부터 받아들인 것을 그대로 쓰거나, 아니면 목적에 맞게 수정을 할 수있습니다.

상속을 하면 한개의 루트 클래스 밑에 여러개의 클래스들을 계층적인 형태로 연관지어 놓을 수 있습니다. 보통 Foundation 프레임워크를 이용해서 만들어지는 코드는, 루트로 NSObject 를

위치시키게 됩니다. Root 클래스를 제외하면, 모든 클래스들은 그 상위로, 즉 루트 클래스에 좀더 가까운 방향으로 하나의 *superclass* 를 가지게 됩니다. 이때 하나의 superclass 라는 점에 주목하십시오. Objective-C 는 다중상속을 지원하지 않습니다. 밑으로는 child class 로 *subclass* 를 가지게 됩니다.

용어를 정리해 봅시다.

C++	Objective-C
Parent class	superclass
Child class	subclass

3.1.1 NSObject 클래스

NSObject 는 root 클래스로써, superclass 를 가지지 않습니다. Cocoa 에서 이 클래스는 모든 클래스의 root 클래스가 됩니다. 이 클래스로부터 상속을하면, runtime 시스템에서 지원을 받는 객체를 만들수 있습니다.

이러한 지원을 받지 않는 클래스를 만드려면 물론 NSObject 로부터 상속을 하지 않아도 됩니다.

3.1.2 인스턴스 변수들을 상속하기

한 클래스를 상속하면 C++의 경우와 마찬가지로 superclass 의 인스턴스 변수들도 모두 다 상속받게 됩니다. 그러므로 NSObject 의 isa 와 같은 인스턴스 변수는 NSObject 를 상속한 모든 클래스에 자동으로 들어가게 됩니다.

3.1.3 method 들을 상속하기

method 역시 인스턴스 변수들과 마찬가지의 방식으로 상속됩니다.

3.1.4 Method overriding

C++과 마찬가지로 Objective-C 도 method overriding 이 됩니다. 즉 superclass 로부터 상속받은 method 를 뭔가 다른 식으로 바꾸고 싶다면, 같은 method 이름을 가진 method 를 subclass 에서 정의함으로써 superclass 의 method 대신에 invoke 되게 할 수 있습니다.

물론 self 와 super 를 이용해서 overriding 된 superclass 의 method 를 호출할 수도 있으니, 그때 그때 상황에 맞게 프로그래밍할 수 있습니다.

단 subclass 는 superclass 의 인스턴스 변수를 overriding 할 수 없습니다. overriding 하는 것은 method 에 한합니다.

3.1.5 Abstract class

이것은 C++의 abstract 와 같습니다. 즉 다른 subclass 들로 하여금 특정 메소드와 인스턴스 변수들을 쓰게끔 하는 효과가 있습니다.

이런 예는 NSObject 가 있겠습니다. 보통 NSObject 의 subclass 를 정의하는 프로그램에선, 그 subclass 들을 사용을 하지 NSObject 클래스로 정의된 인스턴스를 바로 쓰지는 않습니다.

이 abstract 클래스는 abstract superclass 라고도 종종 부릅니다.

3.2 클래스 type

클래스 정의는 어떤 종류의 객체에 대한 spec 입니다. 그러므로 클래스는 어떤 데이터 타입을 정의하는 것입니다. 이때 그 타입은 그 클래스에 정의된 인스턴스 변수같은 데이터 구조뿐 아니라, 메소드와 같은 클래스의 "행위"에 의해서도 정의가 됩니다.

클래스 이름은 C 에서 type specifier 를 쓸 수있다고 한 곳에선 어디든지 쓸 수 있습니다. 즉 예를들면 sizeof 연산자 같은 경우,

```
int i = sizeof( Rectangle );
```

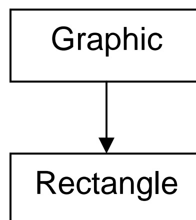
과 같은 식으로 클래스 이름을 sizeof 의 인자로 전달할 수 있습니다.

3.2.1 Static Typing (정적 타이핑)

뒤의 chapter 7. Enabling Static Behaviors 에 나오겠지만, 클래스를 id 와 같은 동적 타이핑이 아닌 static typing 을 이용해서 정의할 수가 있습니다. 이건 마치 C++에서 클래스를 선언하듯이 하는 것입니다. 즉 다음과 같이 합니다.

```
Rectangle *myRect
```

또한 superclass 타입으로 타이핑도 가능합니다. 즉 다음과 같이 상속관계가 있다고 합시다.



그렇다면 다음이 가능합니다.

```
Graphic *myRect;
```

물론 이렇게 선언만 하는 것이야 변수 이름을 마음대로 할 수 있으니 이 예만으론 별 의미가 없겠습니다. 하지만 여기서 말하고자 하는 것은 Graphic 타입으로 선언된 변수가 Rectangle 타입을 실지로 pointing 할 수 있다는 것입니다.

3.2.2 Type Introspection

Type introspection 이란 runtime 시에 해당 인스턴스가 어떤 클래스의 인스턴스인지 등을 알아내는 것을 의미합니다. 좀더 일반적으로 말하자면 runtime 시에 어떤 타입인지, 무엇을 할 수 있는지 등, 실행중에 변할 수 있는 부분을 알아낼 수 있는 메커니즘을 말합니다. NSObject 에는 이런 용도로 isMemberOfClass 와 같은 메소드가 정의되어 있습니다.

```
if( [anObject isMemberOfClass:someClass] )
…
```

라고 한다면 anObject 라는 인스턴스가 someClass 라는 클래스 타입인지 아닌지를 알 수 있습니다. 즉 return 값이 *YES* 이면 someClass 의 타입이란 것입니다.
좀더 포괄적인 것으로는 isKindOfClass 가 있습니다.

```
if( [anObject isMemberOfClass:someClass] )
…
```

이것은 anObject 의 수퍼클래스중 someClass 가 있는가를 알아낼 수있게 합니다. 즉 someClass 의 한 종류인지를 알 수있게 해준다는 것입니다. isMemberOfClass 경우는 직접적인 parent class, 다른 말로는 superclass 의 형인지 아닌지를 알려주는데 반해, 이것은 직접적인 superclass 가 아니어도 된다는 것입니다.

이외에도 주어진 객체가 특정 메시지에 반응하는지 아닌지를 알아내는 것등 introspection 의 예는 많습니다.

3.3 클래스 객체 (Class Object)

클래스를 정의하려면 다음과 같은 정보가 필요하게 됩니다.

- 클래스의 이름과 상속하는 수퍼클래스 이름
- 인스턴스 변수의 template
- 메소드 이름의 선언과, 인자(parameter)와 리턴 타입
- 메소드 구현

즉 C++의 그것과 대동소이합니다.
클래스를 만들때, 컴파일러는 해당 클래스에 대해서 한개의 class object 만을 만듭니다.
이 클래스 객체는 실제로 해당 클래스의 인스턴스는 아닙니다. 인스턴스를 만들기 위한 일종의 템플릿이라고 보시면 됩니다. 하지만 그럼에도 불구하고, 클래스 객체를 위한 특별한 메소드를 가질 수는 있습니다. 이것은 class method 라고 부르는데, instance method 와 개념을 잘 구별하시기 바랍니다. 클래스 객체는 인스턴스 객체와 마찬가지로 그 상위의 클래스로부터 클래스 메소드를 상속받습니다.
소스 코드에서, 클래스 객체는 클래스 이름으로 표현됩니다. 다음 예에서 Rectangle 클래스는 NSObject 클래스에서 상속받은 메소드를 이용해서 버젼 번호를 반환합니다.

```
int versionNumber = [Rectangle version];
```

그런데 클래스 이름이 클래스 객체를 의미할 때는, 메시지 표현에서 receiver 로 쓰였을 때 뿐입니다. 다른 경우에는 인스턴스나 클래스에게 클래스 id 를 반환하라고 요청해야만 합니다. 아래의 두 예는 모두 클래스 메시지에 반응을 합니다.

```
id aClass = [anObject class];
id rectClass = [Rectangle class];
```

이 예제에서 보듯이 클래스 객체도 id 로써 typing 될 수 있습니다. 하지만 클래스 객체라는 것을 더 명확히 보이게 Class 타입으로 나타낼 수 있습니다.

```
Class aClass = [anObject class];
Class rectClass = [Rectangle class];
```

즉 모든 클래스 객체는 Class 타입입니다. 이렇게 선언된 변수는 클래스 이름을 사용하는 것과 마찬가지 입니다. 그러므로 클래스 객체는 다이나믹하게 typing 할 수 있고, 메시지를 받을 수 있고, 다른 클래스로부터 메소드도 상속받을 수 있습니다. 이 타입의 특징은 컴파일러에 의해서 만들어지고, 클래스 정의를 통해서 만들어진 인스턴스 변수를 제외한 어떤 데이터 구조도 가지지 않으며, 런타임시에 인스턴스를 만들기 위한 에이전트라는 것입니다.

NOTE : 각 클래스에 대해서 컴파일러는 메타 클래스 객체라는 것을 만듭니다. 이건 흡사 클래스 객체가 클래스의 인스턴스에 대해서 기술하듯이, 클래스 객체를 기술하는 것입니다. 하지만 이 메타 클래스 객체는 런타임 시스템이 내부적으로만 사용합니다. 즉 여러분은 이 메타 클래스 객체를 쓸 수 없습니다.

3.3.1 인스턴스를 만들기

클래스 객체를 쓰는 주요 이유는 새 인스턴스를 만들기 위함입니다. 다음의 예는 Rectangle 의 인스턴스를 만듭니다.

```
id *myRect;
myRect = [Rectangle alloc];
```

alloc 메소드가 하는 일은, Rectangle 인스턴스의 변수들을 위해서 메모리를 동적으로 할당하는 것입니다. 그리고 또한 0 으로 초기화를 합니다. 단 isa 변수는 만들어진 새 인스턴스를 클래스에 연결시킵니다. 이렇게 만들어진 객체를 쓰려면 완전히 초기화가 되어야합니다. 초기화를 하려면 init 메소드를 호출하면 됩니다. 보통 이것은 allocation 을 하고 바로 하게 됩니다.

```
myRect = [[Rectangle alloc] init];
```

혹은 alloc 과 init 을 따로 할 수도 있습니다.

```
myRect = [Rectangle alloc];
[myRect init];
```

객체는 항상 이렇게 allocation 이 되고 초기화 된 후에 사용해야 합니다. 이에 대한 더 자세한 설명과 Objective-C 에서의 객체를 생성하는 법에 대해서 chapter 10 에 자세히 나와 있으니, 필히 chapter 10 을 참조하시기 바랍니다.

3.3.2 Customization with Objective-C classes

Objective-C 는 OOP 언어이기에 C++과 마찬가지로 상속을 지원합니다. 그럼 도대체 어떤면에서 Objective-C 가 좋다고 할 수 있을까요? 얼핏보기에 대동소이하게 보입니다. 그러므로 굳이 왜 Objective-C 를 써야 하나라고 생각도 듭니다.
뒤에서 나올 protocol 과 category 에 대한 부분에서 더 자세히 알아보겠지만, Objective-C 는 SE 라는 측면에서 상당히 잘 고려되어서 만들어진 언어입니다. 즉 실제 소프트웨어 개발시에 개발자들이 객체를 부품화해서 사용할 수 있도록 되어 있다는 뜻입니다. 물론 C++도 가능하지만 뭔가가 빠져있습니다. 최근에 들어 "OOP 가 소프트웨어 개발에 있어서 무엇을 해 주었는가"라는 자조적인 소리가 컴퓨터 언어 개발과 관련된 사람들이 심심치 않게 이야기하는 데엔 이유가 있는 것입니다.

여기서는 Objective-C 에서 customization 을 어떻게 독특하게 하는지에 대해서 알아보겠습니다. C++에서도 어떻게든 되겠으나, Objective-C 의 메커니즘 때문에 대단히 간단하게 되는 것을 확인할 수 있을 겁니다.

NSMatrix 라는 클래스가 Application Kit 에 있습니다. 이 NSMatrix 를 이용하면 흡사 전자계산기의 숫자판과 같은 것을 행렬판, 혹은 표나 바둑판과 같은 배치를 이용해서 만들수 있습니다. 여기서 중요한 점은 그렇게 만들어진 바둑판의 한 cell 마다 들어갈 실지의 객체들은 버튼이나 텍스트 필드, 아니면 그 외의 무엇이든지 다 될 수 있을것입니다.

보통 C++에서 이것을 접근하게 되면 NSMatrix 를 abstract class 로 만들어 놓고 , 그것을 사용할때는 서브 클래싱을 한다음에, 각각의 바둑판의 한칸을 의미하는 cell 을 만들도록 메소드를 구현할겁니다. 용도에 따라서 각 cell 이 가르켜야 할 객체가 버튼이 될수도, 아니면 그밖의 것이 될수도 있으므로, 이 클래스를 사용하는 사람들은 반드시, 서브 클래싱을 해서 만든 객체가 본인들이 원하는 cell 타입을 지원하는 것인지 알아야만 합니다.
이렇게하면 NSMatrix 에 대해서 뭔가를 항상해 주어야 하고, 한 프로그램에서 여러 타입을 지원하는 NSMatrix 를 만들어야 할 필요가 있으므로, 꽤나 여러개의 NSMatrix 계 클래스를 서브 클래싱을 통해서 만들게 될 겁니다. 이게 뭐랍니까? 더군다나 한 타입을 다루도록 되어 있는 NSMatrix 를 서브 클래싱한 클래스는 그외의 다른 타입을 다룰수가 실질적으로 없습니다.

NOTE : 현재의 C++은 RTTI 를 지원하기 때문에, 이 말은 더 이상 유효하지 않을수도 있다. 또한 abstract 클래스로 cell 을 대표하는 클래스를 abstract class 로 만들고, virtual function 을 만들어서 해결하면 될것이다.
하지만 같은 것을 하기 위해 Objective-C 에서는 얼마나 간단하게 처리를 하는지를 염두에 두어 본다면 여전히 Objective-C 의 장점을 볼 수 있다.

더군다나 비슷한 일을 하는 메소드들을 각 cell 이 가르키는 데이터의 타입마다 다 만들어주어야 합니다.

자.. 여기 좀더 나은 방법이 있습니다. NSMatrix 클래스는 실지로 NSMatrix 객체가 NSCell 이라는 한 타입으로 초기화되게 합니다. setCellClass:라는 메소드를 이용해서 각 cell 을 의미하는 NSCell 에 해당하는 클래스 객체를 전달합니다. 그러면 NSMatrix 는 빈 슬롯을 그렇게 전달된 클래스 객체로 초기화 합니다.

```
[myMatrix setCellClass:[NSButton class]];
```

NSMatrix 는 처음에 초기화 될때 새 cell 들을 전달받은 클래스 객체를 이용합니다. 혹은 크기를 사용자가 바꾸어서 더 많은 cell 을 포함하게 될때도 사용합니다. 만약 클래스가 객체로 취급되지 않았다면 이렇게 하기는 힘들었을겁니다.

이렇게 하면 해결되는 문제는, 앞에서 언급된 방식과 달리 상당히 코드가 간결해 집니다. 또한 비록 NSCell 이 가르키는 객체가 NSButtonCell 이 되었건, NSTextFieldCell 이 되었건, Objective-C 의 특유의 메시징 매커니즘에 따라 다 처리된다는 것입니다.

3.3.3 변수와 클래스 객체(class object)

인스턴스 변수(instance variable)과 클래스 변수(class variable)에 대해서 알아 보겠습니다. 객체에 대한 클래스를 정의할 때, 인스턴스 변수를 만들어 놓을 수 있습니다. 이렇게 만들어진 클래스의 객체들은 각자가 다 이런 인스턴스 변수를 가지고 있게 됩니다.

```
@interface myClass
```

```
{
        int name;
}

myClass classA, classB;
```

즉 이와 같이 하면 classA 와 classB 는 자체 내에 name 이라는 변수를 가지며, 그 내용은 서로 공유가 되지 않는 독립적인 것입니다.

하지만 이런 인스턴스 변수에 대응되는 클래스 변수는 없습니다. 단지 class 정의를 할때 초기화된 내부적인 데이터 스트럭쳐가 있을 뿐입니다. 이런 클래스 객체는 인스턴스 객체의 인스턴스 변수들을 억세스할 수 없습니다. 즉 인스턴스 변수를 초기화 하거나 읽거나 변경을 가할 수 없습니다.
그러므로 인스턴스 객체들이 어떤 데이터를 공유하려면, 그 클래스 외부에 정의된 변수를 써야 합니다. 하지만 이렇게 하면 data encapsulating 의 관점에서 별로 좋지 않을겁니다.

아무튼 이 말이 C++에서의 클래스 변수가 없다는 말은 사실 아닙니다. 단지 instance 변수를 선언하는 것과 대응해서 클래스 변수를 정의하기에 필요한 특별한 type specifier 가 없다는 것입니다. C++에서 클래스 변수를 선언할 때, static 으로 선언하는 것과 마찬가지로 Objective-C 에서도 static 이라고 정의하면 그것이 결국 클래스 변수가 되게 됩니다.

```
@interface myClass
{
        static int example_class_variable;
        int name;
}
```

이렇게 static 타입으로 정의를 하게 되면 그 변수는 해당 클래스 내로 scope 가 제한되며, 그 클래스 내에만 존재한다는 것이 됩니다. 이 *static* 변수는 상속되지 않습니다. 그러므로 모든 인스턴스 객체들은 이 변수의 내용을 공유하게 됩니다.

이런 클래스 변수를 사용하는 용례로는, 만들어지는 인스턴스들의 갯수가 몇개인지등을 기록해 놓는다거나 하는 것이 있겠습니다. 만약 해당 클래스의 객체를 꼭 한개만 만들어야 한다라는 경우에는 그 객체의 모든 상태 정보를 이 클래스 변수들에 넣고, 클래스 메소드만을 이용해서 객체를 만들면 되겠습니다.

3.3.4 클래스 객체를 초기화하기

인스턴스 객체를 할당하기 위해서 사용되지 않는 경우엔, 인스턴스로 초기화 되어야 할 겁니다. 여러분이 만드는 클래스 객체를 할당하진 않더라도, Objective-C 는 프로그램이 클래스 객체를 초기화 할 수 있는 방법을 제공합니다.

즉 클래스가 *static* 이나 global 변수들을 사용한다면, *initialize* 메소드를 이용하면 됩니다.

NOTE : +initialize 메소드는 클래스 메소드로써, 클래스 객체를 초기화할 때 씁니다.
−init 메소드는 인스턴스 메소드로써, 인스턴스 객체를 초기화할 때 씁니다.
앞의 +와 −를 주목하시기 바랍니다.

예를들어 클래스가 자신의 인스턴스들을 관리하기 위해, 만들어지는 인스턴스에 대한 리스트를 가지고 있다면, 그 리스트는 ***initialize*** 메소드에서 초기화하면 됩니다.

runtime 시스템이 ***initialize*** 메시지를 클래스 객체에 보냅니다. 즉 그 클래스가 다른 메시지를 받기 전에, 그리고 그 클래스의 수퍼 클래스가 ***initialize*** 메시지를 받은 후에 runtime 시스템이 보내는 것입니다. 그래서 클래스가 실지로 사용되기전에 초기화를 할 수가 있는 겁니다. 만약 이런 초기화가 필요하지 않다면, 굳이 여러분이 ***initialize*** 메소드를 구현할 필요는 없습니다.

이 역시 클래스 상속에 따른 메시지 전달을 합니다. 즉 어떤 클래스에 대해서 *initialize* 메소드를 구현하지 않았다면, 그 클래스에 대한 ***initialize*** 메시지는 그 수퍼 클래스에 전달됩니다. 비록 그 수퍼클래스가 이미 ***initialize*** 메시지를 받았더라도 말입니다. 그러므로 그 수퍼클래스는 초기화가 단 한번만 수행되도록 만들어져야 합니다.
초기화가 한번만 되게 하려면 다음과 같이 하시기 바랍니다.

```
+ (void)initialize
{
        static BOOL initialized = NO;
        if( !initialized)
        {
                // 초기화를 여기서 합니다.
                …
                initialized = YES;
        }
}
```

NOTE : 여기서 주의할 점은 runtime 시스템이 이 initialize 메시지를 모든 클래스에 보낸다는 것입니다. 즉 클래스 상속을 염두에 두고 현재 만드는 인스턴스에 대해서만 보내는 게 아니라, 한 클래스의 super 클래스에도 보냅니다. 그러므로 여러분이 initialize 메소드를 구현할 때, 그 수퍼클래스에 메시지를 initialization 메시지를 전달하면 안됩니다. 이것은 인스턴스를 초기화할 때 쓰는 init 와는 사뭇 다릅니다.

3.3.5 루트 클래스의 메소드

루트 클래스라 함은 클래스의 상속관계에서 제일 높은 위치, 즉 그 자신의 조상 클래스를 가지지 않는 superclass 입니다.

클래스 객체가 어떤 메시지를 받았을때, 그것을 처리할 수있는 클래스 메소드가 없다면, 런타임 시스템은 그것을 처리해 줄 수 있는 루트 인스턴스 메소드가 있는지를 알아보게 됩니다. 클래스 객체가 수행할 수 있는 유일한 인스턴스 메소드는 루트 클래스에 정의된 메소드 뿐입니다. 단 받은 메시지를 처리해 줄 수 있는 클래스 메소드가 없을때 그렇다는 것입니다.

다시 정리해서 보자면, 받은 메시지를 처리할 클래스 메소드가 없을때, 클래스 객체는 루트 클래스 메소드 중 그것을 처리할 수 있는게 있는지 찾아봐서, 그런 것이 있을때, 그 루트 클래스 메소드를 수행합니다. 즉 상속 관계에서 상위 클래스를 타고 올라가서 최종적으로 root class 의 메소드를 찾게 되는 것입니다.

3.4 소스 코드에서의 클래스 이름

소스 코드에서 클래스 이름은 다음의 두가지 용도로만 사용이 가능합니다.

- 객체를 선언하기 위한 type 이름

  ```
  Rectangle *anObject
  ```

  참고로 인스턴스만이 이렇게 static 하게 typing 될 수 있습니다. 클래스 객체는 이렇게 할 수 없습니다. 클래스 객체는 Class 라는 type specifier 를 이용해서 선언합니다.

- 메시지 표현(message expression)에서 메시지의 receiver 로 사용되어, 클래스 객체를 지칭하기 위해

  바로 이 경우에만 클래스 이름이 클래스 객체를 지칭합니다. 다른 예에서는 클래스 객체가 그 id 를 드러내도록 클래스 메시지를 보내서 클래스 객체를 지칭하는 값을 알아내야 합니다. 다음의 예는 isKindOfClass: 메시지에 Rectangle 클래스를 인자로 전달하는 예입니다.

  ```
  if( [anObject isKindOfClass:[Rectangle class]] )
  …
  ```

  Rectangle 을 인자로 사용하는 것은 안됩니다. 클래스 이름은 메시지의 receiver 로만 사용할 수 있습니다.

만약 컴파일시에 클래스 이름을 알아 낼 수 없고, 런타임시에 그에 대한 스트링을 알고 있다면, *NSClassFromString* 함수가 클래스 객체를 반환해 줄겁니다.

```
NSString *className;
…
```

```
If( [anObject isKindOfClass:NSClassFromString(className)] )
…
```

만약 전달된 클래스 이름이 제대로 된 클래스 이름이 아니라면 nil 을 반환합니다.

클래스 이름은 global 변수와 함수 이름들과 같은 namespace 내에 존재합니다. 그러므로 클래스와 global 변수는 같은 이름을 가질 수 없습니다. Class 이름은 Objective-C 에서 global visibility 를 보이는 유일한 이름입니다.

Chapter 4. 클래스 정의하기

이 부분은 실제로 클래스를 Objective-C 에서 어떻게 만드는지에 대해서 설명합니다. 아마도 제일 앞에 나와야 할 부분이지만, 개념 설명등을 앞에 두어서 부득이 이렇게 뒤에 놓이게 되었습니다. 이것은 Apple 의 도큐먼트 순서와도 일치합니다.

빨리 Objective-C 코딩을 하려면 우선 이 4 장을 먼저 보는 것도 무방할 것입니다.

Objective-C 에서 클래스는 다음의 두 파트를 이용해서 정의합니다

- Interface : 한 클래스의 메소드와 인스턴스 변수를 선언하고, 어느 수퍼클래스로부터 상속을 받는지를 기입합니다. C++의 클래스 구조 정의와 같습니다. 이것은 보통 헤더 파일인 *.h 파일에 정의됩니다.
- Implementation : 실지로 클래스를 정의하는 부분입니다. 즉 메소드를 구현하는 부분입니다. 이것은 C++에서의 멤버 함수를 작성하는 부분과 같습니다. 이것은 보통 구현 파일인 *.m 에 구현됩니다.

컴파일러는 비록 상관을 안하지만, 보통 interface 파일과 implementation 파일을 분리 해두는 것이 좋습니다. 코딩 스타일로도 그렇고, Xcode 에디터도 그런 것을 염두에 두고 "open counterpart"와 같은 기능을 제공합니다. 즉 *.m 과 같은 파일을 가지고 있고, open counterpart 를 하면 거기에 해당하는 헤더 파일이 열려, 클래스의 구조를 파악할 수 있게 해 줍니다. 또한 한 인터페이스 파일엔 한 클래스만을 정의하는 것이 좋습니다. 여러분이 작성하고 있는 프로젝트가 커지면, 한 파일에 여러 클래스를 정의하면 그것을 브라우징하기가 힘들게 될 것입니다.

보통 interface 와 implementation 파일은 그 안에서 구현하고 있는 클래스의 이름을 따서 명명합니다. 즉 Rectangle 이란 클래스를 만든다면 그 구조는 rectangle.h 에, 그 메소드 구현은 rectangle.m 이라는 파일에 해주는 것이 보통입니다.
객체의 인터페이스를 그 구현과 분리하는 것은 OOP 의 패러다임과도 잘 맞습니다. 즉 객체는 그 외부에서 볼때, 그 내부 구현은 보이지 않는 일종의 blackbox 와 같은 독립된 entity 입니다. 그러므로 이렇게 분리하는 것이 좋습니다.

4.1 Interface

클래스를 정의하는 것은 @interface 와 @end 사이에서 합니다. (모든 Objective-C 의 directive 는 @로 시작됩니다.)

```
@interface ClassName : ItsSuperClass
{
        instance variables declaration..
```

```
}

method declarations
@end
```

앞에서 언급했지만 클래스 메소드와 인스턴스 메소드는 다음과 같은 +/- 사인을 앞에 붙입니다.

-	Instance method
+	Class method

이때 인스턴스 메소드와 클래스 메소드의 이름을 똑같은 것을 쓸 수도 있지만, 보통 이렇게 하지는 않습니다. 메소드를 작성하는 것은 다음 형식을 따릅니다.

```
- (return type)MethodName:(parameter_type1)param1Name
                         MethodName_cont:(parameter_type2)param2Name
```

몇가지 예를 살펴봅시다.
우선 인자가 없는, 즉 C 의 경우 void 타입일 경우에는 다음과 같습니다.

```
- (int) doCalc;
```

전달하는 인자가 있을 때는 다음과 같이 합니다.

```
- (void)setRadius: (float)aRadius;
```

타입을 괄호로 둘러 싼다는 점이 특이합니다.
그럼 한개 이상의 인자를 쓸때는 어떻게 할까요?

```
- (void) setWidth: (float) width height: (float) height;
```

복잡해 보입니다. 실제로 이 메소드를 쓸때는 다음과 같겠습니다.

```
id *myRect = [[Rectangle alloc] init];
[myRect setWidth:12.2 height:18.0];
```

즉 전달하는 인자 자체에도 이름이 붙이는 것입니다. 이것은 어떻게 보면 무척 불편합니다. 하지만 이 인자가 뭐였더라하고 자꾸 찾아보는 것이 빈번해진다면 이런 식의 스타일은 상당히 도움이 될 것입니다.

자.. 또 한가지 경우가 더 있죠. 즉 인자의 갯수가 정해지지 않은 경우입니다. 이 경우는 ellipsis 로 나타냅니다.

```
- makeGroup:group, ...;
```

4.1.1 인터페이스 파일을 import 하기

이것은 C/C++에서 #include 를 사용하는 것과 같은 것입니다. 단 Objective-C 에서는 #include 대신에 #import 를 사용합니다.

```
#import <header_name.h>
```

#include 와는 달리 #import 는 해당 해더가 꼭 한번만 include 되게 해줍니다.
흡사 이것은 해당 헤더 파일에 #ifdef.. #endif 나 #pragma once 를 명시해준 것과 같습니다.

사용예는 다음과 같습니다.

```
#import "ItsSuperClass.h"

@interface ClassName : ItsSuperClass
{
        // 인스턴스 변수 선언
}

// 메소드 선언
@endif
```

4.1.2 다른 클래스를 언급하기

앞의 예와 같이 하면 ItsSuperClass.h 에 있는 모든 클래스를 import 하는 것이 됩니다. 만약 이 인터페이스 파일에 선언되지 않은 클래스를 쓰려면, 그 클래스에 대한 인터페이스 파일도 import 하던가 아니면 @class directive 를 써서 어딘가 선언되어 있다고 컴파일러에게 알려주면 됩니다.

```
@class ClassA_name
```

사용예는 다음과 같습니다.

```
@class Rectangle, Circle;
```

이 예에선 Rectangle 과 Circle 이 클래스라고 컴파일러에게 알려주는 것입니다. 이것은 흡사 C/C++에서 extern 을 써서 다른 파일 어딘가에 그에 해당하는 게 있다라고 명시해 두는 것과 같습니다. 이렇게 하면 굳이 해당 클래스의 인터페이스 파일을 import 하지 않아도 됩니다.

하지만 정말 그런 클래스의 메소드를 쓰던가 할때는 반드시 해당 인터페이스 파일을 import 해야 합니다.

이것을 쓸때 얻을 수 있는 장점은, 서로간에 cyclic dependency 를 가지는 경우에 있어도 컴파일 할 수가 있다는 것입니다.
즉 A.h 엔 class A 에 대한 선언이, B.h 엔 class B 에 대한 선언이 있다고 합시다. 그리고 A.m 과 B.m 도 거기에 맞게 되어 있다고 합시다. 근데 서로의 코드에서 서로를 사용하게 되면, 컴파일시에, A 를 컴파일 하려면, B 를 먼저 컴파일해 놔야하고, B 를 하자면 A 를 먼저 해 놓아야 합니다. 그러므로 cyclic dependency 가 생겨 컴파일을 못하게 됩니다. 이때 이 @class directive 를 사용하면 됩니다.

4.1.3 Interface 파일의 용도

Interface 파일은 다음과 같은 용도가 있겠습니다.

- 사용자들이 클래스의 상속관계를 파악할 수 있도록 해준다.
- 컴파일러에게 클래스가 어떤 인스턴스 변수를 가지고 있는지, 어떤 변수를 상속할지 등을 알려준다.
- 메소드를 선언함으로써, 메시지를 처리할 때, receiver 가 그 메시지를 처리할 수 있는지를 파악하게 해준다.

4.2 Implementation

클래스의 구현은 그 선언과 무척 비슷하게 합니다.

```
@implementation ClassName : ItsSuperClass
{
        instance variables declaration..
}

method definition

@end
```

모든 implementation 파일은 그에 대응하는 interface 파일을 import 해야 합니다. 이렇게 함으로써 implementation 파일에서 다음과 같은 것을 생략할 수 있습니다.

- 수퍼 클래스의 이름
- 인스턴스 변수들의 선언

이렇게 interface 파일을 import 하면 위의 선언을 다음과 같이 줄일 수 있습니다.

```
@import “ClasName.h”

@implementation ClassName
```

```
메소드 정의, 즉 구현
@end
```

클래스 메소드는 앞의 4.1 에서 언급했듯이 + 사인을 앞에 붙입니다. 다음의 예를 봅시다.

```
+alloc
{
        ...
}

- (BOOL) isFinished
{
        …
}

- (void) setFilled: (BOOL) flag
{
        …
}
```

또한 클래스 객체의 초기화는 앞의 3.3.4 에서 알아 보았듯이 *initialize* 라는 클래스 메소드를 사용합니다.

```
+ (void)initialize
{
        ...
}
```

갯수가 정해지지 않은 인자를 갖는 메소드는 다음과 같이 정의 합니다.

```
- getGroup:group, …
{
        va_list ap;
        va_start( ap, group);
        …
}
```

Objective-C 에서 특이한 점은 인스턴스 변수 이름과 같은 메소드 이름을 사용할 수 있다는 점입니다.

4.2.1 인스턴스 변수를 억세스하기

클래스의 인스턴스 변수들은 그 클래스의 인스턴스 메소드에서 자유롭게 억세스할 수 있습니다. 컴파일러가 내부적으로 클래스의 구조에 대한 C 스트럭쳐를 만들지만, 프로그래머는 그것을 쓸 필요가 없습니다. 그러므로 .이나 ->같은 스트럭쳐 오퍼레이터를 사용하지 않아도 됩니다. 예를 들어 다음의 메소드는 receiver 의 filled 인스턴스 변수를 사용합니다.

```
- (void) setFilled: (BOOL) flag
{
        filled = flag;
        …
}
```

Receiver 가 아닌 객체에 속하는 인스턴스 변수를 쓸때는, 객체의 타입을 static typing 을 통해서 컴파일러에게 알려주어야 합니다. 이때 ->와 같은 오퍼레이터를 사용할 수 있습니다.
다음의 예를 봅시다.

```
@interface Sibling : NSObject
{
        Sibling *twin;
        int gender;
        struct features *appearance;
}
```

static 하게 typing 된 객체의 인스턴스 변수가 클래스의 scope 내에 있는한, Sibling 메소드는 그 변수를 직접 세팅할 수 있습니다.

```
- makeIdentification
{
        if( !twin)
        {
           twin = [[Sibling alloc] init];
           twin->gender = gender;
           twin->appearance = appearance;
        }

        return twin;
}
```

이건 상당히 특이한 것입니다. 비록 두 다른 인스턴스 객체가 같은 클래스의 형태라고 하더라도 직접 세팅이 가능하다는 것입니다. 그렇다고해서 Objective-C 에 private 같은게 없지는 않습니다.

4.2.2 인스턴스 변수의 scope

C++의 경우와 마찬가지로 3 가지의 directive 가 있습니다.

```
@private
@protected
@public
```

앞에 @가 붙는 것만 제외하면 C++의 경우와 완전히 동일함을 알 수있습니다.

Directive	Scope	상속 가능 여부
@private	선언한 클래스 내에서만	불가
@protected	선언한 클래스와 상속한 클래스	가능
@public	무엇이든 억세스할 수 있음	가능

요새의 C++ .NET 의 새 컴파일러와 달리 이 directive 들이 유효한 범위는 다른 directive 가 쓰일 때까지이다. 즉 C++의 그것과 동일합니다.

```
@private
        int i;
        float k;
                        <= 여기 까지가 private 의 유효 범위이다.
@protected
        int m;
```

만약 이 directive 들을 사용하지 않으면 **default** 는 ***@protected*** 입니다.

	Objective-C	C	C++
Default Directive	@protected	public (for structure & Union)	Private (for class)

NOTE : 위에서 사용한 C/C++의 경우는 Microsoft Visual C++의 MSDN 에서 발췌한 것이다. C 의 경우 Class 란 개념과 이런 억세스 권한이 없는데, 넣어 놓은 것이 이상할 것이다. 기본적으로 Class 는 C 의 structure 에서 그 구현이 확장 된 것으로 볼 수있다. 초기의 C++ 컴파일러들은 사실 C 컴파일러였으며, preprocessor 가 class 들을 C 컴파일러가 이해하는 structure 로 전환한 후에 컴파일을 했다. 그러므로 그 구현이 연장선 상에 있으므로 여기에 명시하였다.

C++의 경우와 달리 super class (혹은 parent class)를 상속 받을 때, access 권한을 명시하지 않는 점에 주목합시다.
(정말로 super class 에 대한 access 권한을 안쓰는 것인지, 아니면 그냥 생략할 수 있는 것인지는 모르겠다. Apple 의 Objective-C 설명서엔 그처럼 자세하게 설명을 해 놓지 않았는데, 아마 처음

Objective-C 를 배우는 사람을 위해서 글을 썼다고 가정한다면, 아마도 이런 기능 자체가 없기 때문이 아닐까 조심스레 추측해 봅니다.)

또 한가지! 클래스 객체는 자신의 인스턴스 변수를 억세스할 수 없습니다.

이런 directive 를 이용해서 C++과 마찬가지의 상속을 할 수 있습니다. 하지만 서브 클래스가 그 수퍼클래스의 인스턴스 변수를 바로 억세스하지 못하게 해야 할 필요가 있을 겁니다. 그런 이유로는 다음과 같은 것들이 있습니다.

- 서브클래스가 그 수퍼클래스의 인스턴스 변수를 억세스하게끔 만들면, 그 순간부터 그 수퍼클래스와 서브 클래스는 뗄레야 뗄 수없게 됩니다. 나중에 변수이름을 바꾸거나 할때 힘들어질 수 있습니다.
- 서브클래스가 그 수퍼클래스의 인스턴스 변수를 바꾸게 되면, 예기치 않은 버그에 노출될 수도 있습니다.

이런 걱정은 사실 C++에선 잘 하지 않는 것입니다. 이것은 respondTo:와 같은 상당히 dynamic 한 메커니즘을 가지는 Objective-C 에서 상속에 의한 것보다 좀 더 나을 수있는 해법을 경우에 따라 제공할 수 있는 Objective-C 의 장점을 최대한 사용하기 위해 고려해 볼만한 걱정이라고 생각이 듭니다.

아무튼 인스턴스 변수를 그것을 선언하는 클래스에서만 쓸 수 있도록 하려면 @private 를 사용하면 됩니다. 이렇게하면 서브 클래스에서는 수퍼 클래스의 메소드만을 이용해서 억세스할 수 있게 됩니다. 어디서든 억세스하게 하려면 @public 으로 선언하면 됩니다.

```
Worker *ceo = [[Worker alloc] init];
ceo->boss = nil; // boss 는 Worker 클래스에서 @public 으로 선언된 것입니다.
```

이때 ceo 는 위의 예처럼 static 하게 typing 되어야 합니다.

Chapter 5. How Messaging Works

Objective-C 언어는 OOP 언어이다. 이 언어 역시 entry point 는 main() 함수이므로 어느 정도의 procedural 한 면이 있으며, 여전히 procedural 하게 코딩을 할 수 있다. 하지만 Apple 이나 GNU 의 Objective-C 문서를 보면, 그리고 Interface Builder 에 의해서 만들어지는 코드와 그에 기반한 개발 프로세스를 보면 이런 procedural 한 면이 Visual C++의 MFC 나 C++ 자체의 그것에 비해 많이 숨겨짐을 알 수 있다.

즉 프로그래밍 모델을 Object 들의 존재와 그들 사이의 통신으로 몰아간다.

필자가 보기에 여기에 Objective-C 에서 messaging 이란 단어가 많이 사용되는지 이유가 있다고 본다.

C/C++ 프로그래머에게 message 가 뭔지 쉽게 설명하자면 그저 C++ 클래스의 멤버 함수, 즉 method 이다. 아마도 messaging 이란 이름을 사용하는 이유는 OOP 모델에서 프로그래밍은 오브젝트간의 method call 이기 때문이 아닐까 생각한다. 이렇게 생각하면 왜 message 라고 하는지 이해가 간다.

***참고** : C++ 책이 아닌 순수 OOP 책들을 보면, member function 이라는 C++의 용어보다는 메시지란 말을 쓴다. SmallTalk 의 영향이 강한 것 같다.

Objective-C 로 이것을 보면 다음과 같다.

```
[myClass setName:my_class_name_string];
```

여기서 Objective-C 의 중요한 개념 중 하나인 selector 가 나온다. Objective-C 는 대단히 dynamic 한 언어이므로 메시지를 받게 될 객체의 type 이 여러 개일 수 있다. 그중 그 메시지에 해당하는 method 는 각 객체마다 다 다르게 구현되어 있을 수 있다. Objective-C 의 run-time 은 그런 것을 구별해 주어야 적절하게 메시지들 전달할 수 있다. 그렇게 비슷하지만 다른 method 들을 선택할 수 있게 해주는 개념으로 selector 가 나왔다. 이것은 C/C++ 프로그래머의 입장에서 보면 function pointer 를 이용해서 그때 그때 상황에 맞는 method/function 을 호출하는 것과 비슷하다.

method selector 들은 dispatch table 이란 곳에 저장이 된다. run time 이 적절한 함수를 찾기 위해 이 dispatch table 을 이용하는데, 이때 각 class 의 isa pointer 를 따라서 그 테이블을 traverse 하게 된다. 그러므로 여러분이 class 를 정의할 때, isa pointer 를 갖도록하는게 중요하다. 참고로 NSObject 나 NSProxy 로부터 상속을 받으면 자동으로 이 isa pointer 를 가지게 된다.

더 자세한 내용은 Apple 의 Objective-C 에 대한 공식 문서를 참고하시기 바랍니다.

5.1 Selector

컴파일된 selector 는 SEL 이라는 특별한 타입으로 되어 있다.

Selector 를 강제로 설정한다거나, 혹은 따오려면 다음과 같이 한다.

```
SEL setWidthHeight;
setWidthHeight = @selector(setWidth:height:);
```

흡사 C/C++ 프로그래머에겐 setWidth:height:라는 함수의 어드레스를 가져와서 setWidthHeight 란 함수 포인터에 저장하는 것과 유사하게 보인다.
(실제 구현이 어떤지는 모르겠다.)

class method 와 instance method 는 같은 이름을 가질 수 있고, 그때 selector 도 같다. 하지만 두개 사이의 confusion 은 없다.

dynamic binding 이 되는 경우에 같은 selector 를 가지는 method 들은 같은 return type 과 같은 패러미터를 가져야 한다. static binding 이 되는 경우는 그렇지 않아도 된다.

이 selector 를 어떻게 쓰는지 한번 보자.

```
[friend gossipAbout:aNeighbor];
```

위의 문장은 friend 라는 object 의 gossipAbout 이라는 method 를 call 하는데 패러미터로 aNeighbor 를 준 것이다.
이것을 selector 를 써서 구현하면 다음과 같다.

```
[friend performSelector:@selector(gossipAbout:)
        withObject:aNeighbor];
```

즉 friend 라는 객체에서 gossipAbout: method 의 selector 를 따다가 수행을 시키는데, aNeighbor 라는 객체를 가지고서 해당 method 를 수행한다.

여기서 느껴지는 것은, 첫째 뭔가 run-time 시에 호출할 함수를 결정짓는다는 느낌이 강하다.
둘째, 정말이지 selector 는 함수 포인터같다는 점이다.

Cocoa 프레임워크는 이 selector 를 무척이나 많이 사용한다. C/C++ 개발자에게 이 selector 는 처음엔 무척이나 당황스러운 기능이다. 적어도 필자는 그러했다. -..-
근데 가만히 보니 C/C++로 callback 함수를 만들고 전달해줄 때, 결국 invoke 될 함수의 이름을 패러미터로 전달해주는 것과 많이 닮았다. 함수의 이름은 그 함수의 pointer 로 작용한다는 점을 생각하면 이 selector 를 쓰는 것과 그런 callback 함수를 사용하는 것과 대단히 흡사하다.

그러므로 대부분의 경우에 selector 라는 말을 들으면, "아.. 함수 포인터" 이렇게 생각하는게 이해가 빠를 것 같다.

아무튼 Objective-C 에서의 method overiding 이나 overloading 은 이 selector 메커니즘을 이용해서 한다는 생각이 든다.

그리고 이 점은 Objective-C 가 C 의 진정한 superset 이고 어떻게 OOP 를 C 에 접목시키면서 그렇게 단순한 모양을 가지게 되었는지 알 수있게 해준다.
사실 이런 부분에선 C++처럼 컴파일러를 아예 새로 만들어야 하는 접근이 더 나아 보이기도 한다. 하지만 C++은 그러다보니 복잡성 또한 증대 되었다.
그리고 이런한 접근법의 차이가 OOP 프로그래밍 스타일의 차이도 불러온 것 같다.
Objective-C 로 만든 코드를 보면, 오브젝트간의 메시지 교환들이 모여 하나의 프로그램이 되는 느낌이 강한 반면, C++ 코드를 보면 언어자체는 애초부터 OOP 를 지원하지만 그 코드는 웬지 procedural 한 코드에 객체가 더해진 것 같은 느낌이 든다.
이 점을 인지하는게 필자의 경우엔 C/C++ 코딩에서 Objective-C 코딩에 익숙해지는데 핵심이었다. 이게 눈에 보이니까 비로소 Objective-C 코드가 보이기 시작했다.

5.1.1 Messaging Error 처리

무슨 말인가하면 어떤 객체의 method 를 dynamic 하게 호출했다고 하자. 앞에서 살펴본 selector 를 이용해서 일종의 함수 overriding 과 overloading 을 하는 셈인데, 만약 그 메시지를 받는 객체가 해당 함수를 가지지 못할 때의 문제를 말하고자 하는 것이다.
즉 이건 뭐랑 같은가 하면 C++에서 어떤 클래스에 존재하지 않는 method 를 호출한 셈이다. static typing 을 쓰면, 즉 어떤 객체등을 사용할때, 그 타입으로 선언을 하고 쓰면 Objective-C 컴파일러가 컴파일시에 에러를 내 주지만, 만약 그 타입이 id 와 같은 dynamic type 이었고, run time 시에 어떤 객체로 일을 하게 될지 결정하게 된다면 문제가 될 것이다. 존재하지 않는 함수를 호출한다...

Objective-C 는 여기에 대한 해결법을 가지고 있다.
다음 소스를 보자.

```
if ( [anObject respondsToSelector:@selector(setOrigin::)] )
        [anObject setOrigin:0.0 :0.0];
else
        fprintf(stderr, "%s can't be placed\n",
                        [NSStringFromClass([anObject class]) cString]);
```

respondsToSelector 는 *setOrigin* 이란 메시지를 *anObject* 가 알아 먹는지.. 즉 *setOrigin* 이란 method 를 *anObject* 가 가지고 있는지 run time 시에 알아내주는 함수이다. 위의 소스에서 보자면, 그것에 반응을 한다면 즉 setOrigin 메소드를 *anObject* 가 가지고 있다면 *anObject* 의 *setOrigin* 을 호출하고, 그렇지 않으면 *anObject* 의 클래스가 어떤 것인지 그 이름을 출력해준다. (이 이름을 출력해주는 부분도 참 강력한 부분이다. 동적으로 현재의 클래스 이름을 파악할 수 있게 해준다.)

자 여기서 하나 더 생각해 보자. 주어진 object 가 해당 메시지를 처리할 수 없다고 하자. 그럼 어쩔 것인가? 경우에 따라서 "나 그거 해 줄 능력없어" 하고 에러 메시지내고 빠져 나갈 수도 있고, 혹은 그

능력을 가진 넘에게 토스를 해 줄 수 있다. 이 개념이 실제로 Objective-C 에 존재하는데 그것을 "forwarding"이라고 한다.
이에 대해선 뒤에서 알아보기로 하자.

5.2 Hidden Argument

Objective-C 클래스의 method 에는 두개의 숨겨진 인자가 있다. 바로 self 와 _cmd 이다. self 는 C++의 this 와 마찬가지로 자기 자신의 객체를 지칭하고 있으며 _cmd 는 그 호출된 method 이름을 가지고 있다. 더 정확하게 말하자면 해당 method 가 호출될 때, 사용된 selector 를 이 _cmd 가 가지고 있게 되는데, 결국 그 selector 가 선택한 함수는 그 호출된 method 이다. 그러므로 _cmd 는 호출된 method 의 selector 를 가지고 있는 것이다.

이에 대한 소스코드를 한번 보자.

```
- strange
{
        id target = getTheReceiver();
        SEL method = getTheMethod();
        if ( target == self || method == _cmd )
        return nil;
        return [target performSelector:method];
}
```

단 여기서 주의할 점은 self 는 C++의 this 처럼 static 하지 않다는 것이다. 실제로 message 를 받은 객체의 포인터이다. (결국 비슷한건가?)

NOTE : XCode 의 디버거는 이 self 와 _cmd 를 보여줍니다.

5.3 Message to self and super

Objective-C 에서의 self 는 C++에서의 this 와, super 는 C++의 super 와 같다. 그러므로 새로울 것은 없는데, 단 코딩 스타일에 있어서 염두에 두어야 할 것이 있다.

C++에서 혼돈할 여지가 없는 한, 그냥 memeber function 이나 그 객체의 variable 을 쓸 수 있다. 그 앞에 this->를 붙여도 상관없지만, 대개 this 는 컴파일러가 혼돈을 일으킬 소지가 있을때 사용한다. 하지만 Objective-C 에서 메시지 invocation 은

```
[theObject theMethod]
```

의 형식을 가지기 때문에 반드시 self 를 쓰게 된다.

super 또한 그 용례에 있어서 차이점이 있다. C++에도 super::가 있어서, 예를 들어 parent 클래스의 member function 을 invocation 하려면

```
super::theFunction()
```

과 같은 식으로 한다. 하지만 MFC 등의 코딩 스타일을 보면 super 를 사용하는 것보다는 그 parent class 의 이름을 직접 거명해서 하는 경우가 더 흔하다. 즉 super 의 위치에 그 parent class 의 이름을 직접 쓰는 것이다.
아마 이는 super 라는 이름이 C++에서 나온게 아니고 최근에 첨가된 이유때문인지도 모른다.

참고로 Simular school 과 SmallTalk school 은 같은 것에 대해서 명명하는 것이 좀 틀리다. 정리 해 보면

	C++ (Simular)	**Objective-C (SmallTalk)**
class 의 변수	member variable	property
class 의 함수	member function	method
parent class	parent class	super class
class 자신	this	self

그런데 self 는 this 와 큰 차이점이 있다. Class method 에서 쓰이면 self 는 class 를 가르키며, instance method 에서 쓰이면 self 는 그 instance 를 가르키게 된다.

이외에도 self 에는 중요한 이슈가 있으니, 이는 Apple 의 Objective-C 문서를 참고하기 바란다.

Chapter 6. How to Extend Classes

Objective-C 도 OOP 언어인 만큼 C++과 마찬가지로 Polymorphism 을 지원한다. 하지만 그것을 비슷하면서도 사뭇 다르게 지원한다. OOP 의 관점에서, 그리고 Software Engineering 관점에서 보면 Objective-C 는 C++에 비해 확실히 잇점을 제공한다.
그것은 두가지의 개념 즉, Category 와 Protocol 이란 개념으로 가능해진다.

6.1 Category

카테고리란 개념은 이미 있는 클래스에 method 들을 추가할 수있게 해준다. 또한 클래스 정의들을 구분해준다. 처음엔 이 말이 잘 이해가 안된다. 이 중 특히 주의를 두어야 할 부분이 "이미" 있는 클래스에 method 들을 추가할 수 있게 해준다는 부분이다.

이것은 C++의 inheritance 와 사뭇 다르다. 자 A 라는 프로그래머가 B 라는 사람으로 부터 ClassB 에 대한 헤더와 object file 을 받았다고 하자. A 라는 프로그래머가 작업을 하다가 보니까 ClassB 에 기능을 추가할 일이 생긴 것이다. 만약 이 프로그래들이 C++로 작업을 하고 있다면, ClassB 에 method 를 추가할 수 있는 유일한 방법은 ClassB 를 상속받아서 ClassA 란 이름으로 sub class 를 만들고 거기에 새 member function 을 정의하는 것이다. 상당히 수직적이다.

자 이제 Objective-C 를 이용한다고 해보자. 물론 Objective-C 도 inheritance 를 지원하는 만큼 C++처럼 할 수도 있다. (단 다중 상속은 안된다. 즉 복수개의 parent class 를 동시에 상속 받을 수 없다는 말이다. 아.. 이말은 상속의 상속을 받을 수 없다는 뜻이 아니다. 어떤 한 레벨에서 동시에 복수개의 parent class 로부터 상속 받을 수 없다는 뜻이다. 그러므로 특별히 상속의 강력함에서 C++에 비해 뒤지지 않는다. 많은 경우에 있어서 다중 상속은 잇점보다는 해악(?)이 많다는 것을 생각해 보자.)
아무튼 본론으로 돌아가서 Objective-C 는 이러한 수직적인 상속외에, 기존의 클래스를 확장하는 방법으로, 수평적 방법을 사용할 수 있다.
그것을 가능하게 해 주는 것이 바로 Category 이다. 즉 Category 를 이용하면 class 의 층을 만드는 것을 줄일수 있다. 요새 라자냐코드라는 말을 많이들 한다. 라자냐코드란 불필요하게 클래스 상속을 많이 받아서 클래스 층이 많아진 것을 말한다. procedural 모델에서 스파게티 코드와 마찬가지로 이 라쟈냐 코드도 디버깅시에 많은 골치를 안겨주며, 불필요하게 수행시 메모리를 많이 잡아먹게 하는 악행을 저지른다. 또한 이 둘 사이의 공통점은.. 둘다 음식의 이름에서 유래되었다.

어떻게 하는지 코드를 보자. 우선 헤더 파일을 보면

```
#import "ClassName.h"
@interface ClassName ( CategoryName )
        method declarations
@end
```

그리고 implementation 파일을 보면

```
#import "CategoryName.h"
@implementation ClassName ( CategoryName )
        method definitions
@end
```

형식은 애초에 클래스를 정의하고 만드는 것과 다르지 않다. 단 하나 차이가 있다면 뒤에 수상하게 생긴 괄호로 된 부분이 있다는 것이다. 바로 이 부분이 카테고리에 관련된 것이다. 즉 앞의 ClassName 은 이미 있는 클래스의 이름이고, 여기에 새로운 method 를 추가할 때, 카테고리 이름을 뒤에 괄호로 해서 넣고, method 를 추가하는 것이다. 그럼 sub classing 을 하는 것과 달리 수평적으로 해당 클래스를 확장하게 된다. 즉 ClassName 이라는 클래스에 새로운 method 가 추가된다.

여기서 하나 짚고 넘어갈 점은 Category 로는 기존의 클래스에 instance variable 을 넣을 수 없다는 것이다. 오직 method 만을 추가할 수 있다.
또 하나의 중요한 점은 같은 클래스를 확장하는데 다른 카테고리 이름을 줄 수가 있다는 점이다. 이게 어떤 효과를 발휘하는가?
역시 C++과 비교를 해 보자.

ClassC 라는 애초의 클래스가 있고 그 child 로 ClassB 가 있다고 하자. 역시 A 라는 프로그래머가 B 라는 프로그래머에게 ClassB 를 받았다. 왜냐하면 B 라는 프로그래머는 C 라는 프로그래머에게 ClassC 와 관련된 파일을 받아서 그걸 작업했기 때문이다. B 는 A 에게 ClassC 와 관련한 파일도 줄 수 있지만 어차피 자기가 그것을 상속받아서 ClassB 를 만들었기 때문에 굳이 그럴 필요가 없다고 생각되어서 ClassB 에 관한 것만 A 에게 준 것이다.
자.. 이 경우 A 라는 프로그래머 입장에서 기존의 클래스와 관련된 새 클래스를 만들고 싶은데, ClassB 에 있는 새로 추가된 기능은 필요하지 않다고치자. 이럴 경우에 어쩔 수 없이 A 는 ClassB 를 상속받아서 자기가 필요한 method 를 넣어 ClassA 를 만들 것이다. 아.. 문제다. 실행엔 문제가 없지만 쓸데없는 ClassB 의 method 가 항상 ClassA 를 만들때 따라 다닌다. 수행되고 있는 코드의 크기 증가를 의미한다.

자 이제 Objective-C 를 보자. B 가 ClassC 를 확장하는데 Category 를 사용했고 그 카테고리 이름을 CategoryB 라고 지었다고 하자. 그리고 A 는 ClassC 를 확장하는데 CategoryA 라고 이름을 지었다고 하자. 수평적 확장으므로 레이어가 ClassA 에서 두개밖에 없을 뿐 아니라, B 가 추가한 부분이 A 가 확장한 ClassC 클래스엔 들어 있지 않다!
아.. 얼마나 좋은가? 멋지지 않은가?

자.. 이제 C++의 관점에서 다시 보자. 그렇다면 애초에 B 가 A 에게 ClassC 에 대한 파일도 주면 될게 아닌가? 그렇긴하다. 그러므로 엄밀하게 말해서 굳이 Objective-C 가 C++에 비해서 이런 면에서 꼭 좋은 것은 아니다. 하지만 실제 상황을 보자.
Objective-C 는 NeXT 가 Software Engineering 을 고민하면서 채택하게 된 언어다. 즉 현장에서 프로그램을 개발할때 발생하는 문제를 고려했다는 이야기다.

자.. A 라는 사람이 일하는 회사는 CompanyA 다. 같은 식으로 B 는 CompanyB, C 는 CompanyC 에서 일한다. CompanyB 에선 CompanyC 가 만든 라이브러리를 사서 개발을 했다.
하지만 CompanyA 는 CompanyB 의 라이브러리만을 샀다. 그러므로 CompanyA 는 CompanyC 의 라이브러리에 정의된 것을 바로 쓰지 못한다.
이럴때.... C++의 모델을 보자. 아마 십중팔구 ClassB 의 필요없는 부분을 안고 가야할거다. 왜냐하면 A 는 B 가 제공해준 헤더와 오브젝트만을 가지고 일하기 때문이다.
자.. 이제 Objective-C 를 보자. 이 경우엔 비록 B 로부터 필요한 것을 받아도, B 의 추가 부분을 안고가지 않을 수가 있다! 또한 classC 의 private 한 부분도 그냥 막 쓸 수 있다.
상속 받은게 아니라 ClassC 에 직접 method 를 넣은 것이니까!

SE 측면에서 상당히 좋은 것이다. 이건..

Objective-C 의 장점.. 컴포넌트웨어를 가능하게 해 주었던 부분이다.
NeXTStep 당시에 오브젝트를 만들어서 팔던, 그래서 조립하는 식으로 프로그래밍이 가능했던 이유 중의 하나가 여기에 있다.

여기서 하나 짚고 넘어가자. 카테고리 안에 정의된 method 는 ClassC 의 원래 method 의 이름과 같은게 있을 수도 있다. 이때는 기존의 것을 override 하는 것이 된다!
하지만 이때는 상속 받은 것과 달리 기존의 것을 새로운 것이 invocation 을 할 수가 없다.
그냥 완전히 새걸로 바꾸는 것이다. 이것은 장점이기도 하고 경우에 따라 단점이기도 하다.

또한 같은 클래스에 대한 다른 카테고리에 있는 것을 override 하는 것은 안된다.

자.. 하지만 기존의 클래스에 변수를 추가해야 할 경우엔 어쩔까?
이럴 경우엔 어쩔 수없이 상속을 하면 되겠다. 여기에 대한 카테고리 비스무레한 메커니즘이 있음 좋겠다 싶지만, 현재로썬 이렇다.
아무튼 이렇게 함으로써 필요에 맞게 상속이라는 수직적 클래스 확대와 category 라는 수평적 확대를 필요에 따라 적절히 쓸 수 있다. 이런 Minimalistic 한 접근도 objective-C 의 큰 장점이라 하겠다. 즉 기존 메커니즘을 이용하면서 꼭 필요한 새 기능을 최소화해서 넣는 것이다. 아마 이런 minimalistic 한 사고 방식때문에 Objective-C 는 C++처럼 비대해지지 않았을런지도 모른다. (물론 C++엔 template 이라던가, operator overloading 과 같은 새로운 개능도 많다. 이런 것은 Objective-C 도 받아 들였음 하는데, 현재는 gcc-objc 가 C++도 컴파일 하기 때문에, C++ 코드와 섞어 쓸 수있어서 어느 정도는 해결이 된다고 본다. 하지만 섞어 쓰는 것과 Objective-C 자체에 이런 새 기능을 넣는 것은 차이가 있다.

6.1.1 Category in Root Classes

Root 클래스라 함은 상속 관계에서 가장 부모 node 의 클래스라는 것이다.
이 경우는 두가지 고려할 점이 있다. Objective-C 에서 Root class 쪽은 항상 Class 냐 class 의 instance 인 object 이냐를 생각해줘야 한다는 점을 염두에 두고 생각하면 쉽다.

- super 로 메시지를 보낼 수 없다. (이미 자기가 root 이므로 더 이상 super class 가 없다.)
- class object 는 root class 에 정의된 instance method 를 억세스할 수 있다.
(보통은 class 오브젝트는 instance method 를 실행할 수 없지만 root class 의 경우엔 예외이다.)

6.2 Protocol

class 정의와 category 와는 달리 protocol 은 특정 class 의 메소드 정의와 별로 상관이 없다. 대신 다른 class 들이 아마도 구현하게 될 method 나 혹은 class 들은 구현하지 않을 method 를 미리 명시해 두는 것이다.
후자를 제외하고 보면 이것은 흡사 C++의 friend 와 비슷한 면이 있다. 하지만 사뭇다르며 C++의 friend 가 할 수 있는 것보다 훨씬 더 넓은 개념이다.

이제 깔끔하게 다시 protocol 에 대해서 정의하자면,

"클래스랑 관련 없는, method 선언의 모음"

이다.

만약 protocol 로 선언된 method 를 써야 할 필요가 있다면, 그 클래스는 해당 protocol 을 쓰겠다고 하고(adopt), 그 method 를 자체 내에서 구현하면 된다.
보통 protocol은 관련있는 메소드들 끼리 모아 놓게 된다. 그러므로 어떤 클래스가 그 protocol을 사용하겠다고 하면, 그 클래스는 그 "통신 규약"[1]을 이해하는 클래스가 된다.

이 protocol 이 대체 어디에 도움이 되는지 보자.
protocol 로써 declare 된 함수들은 구현이 되건 안되건, 어떤 클래스에 구현이 되건 말건이 주 관심사가 아니다. 그보다는 어떤 클래스가 어떤 protocol 을 이해하는지, 즉 conformity 에 관심을 둔다.
즉 예를 들어 마우스 이벤트에 대한 함수를 protocol 로 만들어 두었다고 하자. 그리고 어떤 클래스가 그 protocol 을 쓰겠다고 하자. 그럼 그 class 는 그 마우스 이벤트를 이해하는 클래스가 되는 것이다. 아마 이래서 protocol 이란 이름을 붙인 것으로 생각된다.

이것은 dynamic language 의 입장에서 볼때, class 를 inheritance 와 category 에 의거한 "어디에서 상속을 받았나"로 클래스들을 구분해 놓는 것 뿐만 아니라, protocol 로 인해서 어떤 클래스들이 서로 비슷한 행동을 하나라는 동적인 구분까지 가능하게 해 준다.
개인적으로 이 부분은 메시지 delegation 혹은 forwarding과 더불어, 메시지를 받는 object가 무엇이 되었건 해당 메시지를 처리할 수있도록 하는 메커니즘을 제공한다고 본다. 비교적 static한 언어인 C/C++에서 이런 것을 처리하려고 한다면 void 포인터나 parent class의 포인터를 이용해서

[1] protocol을 단지 용어의 의미가 아니라, 왜 protocol이란 이름이 붙었는지를 강조하기 위해서 "통신 규약"이라고 바꾸어 불렀다.

처리해 줄 수 있지만, 경우에 따라서 아예 클래스의 종류가 다른 것에 대해선 어찌해 줄 방도가 없다.[2] 하지만 Objective-C는 그런것까지 가능하게 해준다.

언제 protocol 을 사용하게 되는지 정리해 보자.

- 다른 프로그래머들이 구현하게 될 메소드의 선언 (Apple 문서에서 따온 말)
  하지만 이 말보다는 "다른 프로그래머들에게 구현하라고 유도해야 할 메소드들의 선언"이라고 하는 편이 더 맞겠다.

- 오브젝트의 클래스 자체는 숨기면서 그에 대한 인터페이스를 만들어야 할 때.

- 계층적으로는 전혀 상관 없는 클래스들이지만 뭔가 상호연관성을 가지게 해야 할 필요가 있을때.

Apple 의 문서에서는 protocol 을, 정의되어 있지 않은 객체에 메시지를 전달하고자 할 때 쓸 수 있다라고 되어 있다. 즉 아직 만들지 않은, 혹은 다른 사람이 작업하고 있어서, 그 클래스의 구조를 정확하게 모를때, 그 클래스에 어떤 메시지를 전달하는 여러분의 루틴을 만들 수 있다는 말이다. 그렇게 생각할 수도 있겠다. 이럴 경우, 대상이 되는 객체에 대한 정의 파일인 인터페이스 파일을 가지고 있지 않는 경우다. C++의 경우는 꼭 정의를 가지고 있어야지만 하지만 Objective-C 는 이렇듯 대상에 대한 정보가 없어도 되는 메커니즘을 가지고 있다.

구체적으로 어떻게 쓰는지 Apple 의 문서를 참조해 보자.

어떤 한 객체 A 가 있는데, 그것이 helpout 이라는 메시지를 다른 객체 B 에 보내서 어떤 일을 수행해 달라고 요청한다고 하자. 이때 현재 코딩하고 있는 객체 A 에 다음과 같은 함수를 이용해서 객체 B 를 기록해 놓을 수 있겠다.

```
- setAssistant:anObject
{
        assistant = anObject;
}
```

그리고 이 객체에 메시지를 보낼때는, 그 메시지를 처리하는 측에선 자신이 그 메시지를 처리할 수 있는지를 검사하는 루틴을 넣어야겠다.

```
- (BOOL)doWork
{
        ...
        if ( [assistant respondsTo:@selector(helpOut:)] )
        {
                [assistant helpOut:self];
```

helpOut 이라는 메시지에 해당 객체, 즉 assistant 가 가르키고 있는 객체가 반응을 하는지를 알 수 있다.

[2] 이 부분은 현재의 C++에선 좀 거리가 있는 이야기다. 현재 C++은 RTTI를 지원함으로써, 부분적으로 이런 것을 동적으로 처리할 수 있게 해준다.

```
                return YES;
            }

            return NO;
        }
```

6.2.1 Protocol 이용 : 익명의 객체를 위한 Interface 를 선언하기

protocol 을 이용하면 익명의 객체(anonymous object)가 가지게 될 method 를 정의할 수 있다. 익명의 객체라고 함은 아직 알지 못하는 클래스의 객체이다. 이런 객체는 서비스나, 몇몇개의 함수들에 대한 handle 도 될 수 있다. 특히 그런 것들에 대해서 한 객체만이 필요할 때 그런 handle 을 쓸 수 있다. 단 프로그램의 구조을 정의하는데 아주 기본적인 역할을 하는 것이나, 사용하기 전에 반드시 초기화해야만 하는 그런 것은 이런 익명의 객체엔 적합하지 않겠다. 아무튼 이까지 읽고 보면, 이것은 사뭇 C++의 template 과 같은 느낌이 든다. 사실 void pointer 에 더 가까운 느낌이 들지만, 웬지 template 이 할 수 있는 것을 할 수 있을 것 같은 생각이 든다.

언제 이런 익명의 객체가 개입될 수 있는지 정리해 보자. 우선 해당 객체를 만드는 개발자에겐 익명의 객체란게 있을 수는 없겠다. 하지만 그 개발자가 자신이 만든 라이브러리, 혹은 framework 나 여러 객체를 다른 개발자가 쓰도록 하는 상황이라면 좀 달라질 수 있겠다.

- 라이브러리나 framework 제작자는 클래스 이름이나 인터페이스 파일로는 알 수 없는 객체를 만들수 있다. 그런 이름이나 클래스 인터페이스가 없다면, 그런 것을 가져다 사용하는 사람들 입장에선 도무지 그런 클래스의 instance 를 만들래야 만들수가 없겠다. 아니면 원제작자는 이미 만들어 놓은 instance 를 제공해 주어야만 한다. 보통의 경우, 다른 클래스에 정의된 method 는 사용할 수 있는 객체를 리턴해주게 끔 만든다.

  ```
  id formatter = [receiver formattingService];
  ```

  이 예에 있는 객체, 즉 반환된 객체는 그 class identity 가 없다. 아니면 적어도 그 원 제작자가 바깥에 공개하기를 꺼리는 것일 거다. 이런 객체를 쓰게 하려면, 적어도 몇몇개의 메시지에는 반응을 하도록 만들어야 한다. 그러므로 이런 것은 protocol 로써 정의된 method 들과 연결시켜 놓을 수 있다.

- Objective-C 메시지를 원격 객체(remote object), 즉 다른 프로그램에 있는 객체에 보낼 수 있다.

  각 프로그램들은 각자의 구조와, 클래스, 그리고 동작하는 논리가 있지만, 다른 프로그램이 어떻게 움직일지 여러분이 꼭 알아야 하는 것은 아니다. 그 프로그램의 바깥에 있는 사람 입장에선, 어떤 메시지를 보낼 수 있고, 어디다 보내야 할지를 알면 그만이다. 이것은 결국 protocol 과 receiver 이 무엇이냐를 알기만하면 된다는 것이다.

> 다른 곳에서 오는 메시지를 받아 처리할 객체를 공개하는 프로그램은, 반드시 그 메시지에 반응할 method 또한 공개해야 한다. 메시지를 보내는 측에선 해당 객체의 클래스가 뭐냐에 대해선 알 필요가 없고, 그 클래스를 내부적으로 이용할 필요도 없다. 단지 필요한 것은 protocol 뿐이다.

이와 같이 protocol 로 말미암아 익명의 객체라는 것을 만들 수가 있다. 만약 이런 protocol 이 없다면 클래스가 뭔지를 알지 못한다면, 그 객체를 만들수도 없다.

NOTE : 익명의 객체를 공급하는 측이 그 class 를 밝히지 않을지라도, runtime 시에 객체가 자신의 class 를 공개하긴한다. class 메시지가 그 익명의 객체의 class 를 반환하게 한다. 하지만 이런 정보를 굳이 안다해서 과히 도움될 것은 없겠다. 그 이유는 protocol 자체에 있는 정보만으로도 충분하기 때문이다.

6.2.2 Protocol 이용 : 비계층적 유사성(Non-Hierarchical Similarities)

여러 클래스가 같은 method 를 구현하게 된다면, 그런 클래스들은 한 abstract class 밑으로 들어갈 수 있으며, 그 abstract class 에서 공통된 method 들을 선언해 놓게 된다. 각각의 sub class 에선 그런 method 들을 각각의 용도에 맞게 구현하면 된다. 이렇게 함으로써 그런 클래스들은 상속에 의한 계층적 구조에서, 그리고 같은 abstract class 를 그 조상으로 한다는 점에서 유사성을 가지게 된다.

하지만 때때로 abstract class 에 공통적인 method 를 다 잡아 넣을 수 없는 경우도 있다. 이를테면, 전혀 관련이 없는 클래스들이지만 유사한 method 를 가지고 있을 수도 있는 것이다. 이럴 경우에 상속관계로 묶어 놓기엔 좀 곤란할 수 있을 것이다. 예를 들어, 완전히 다른 클래스들에 reference counting 을 위해서 어떤 method 를 구현해 넣는다던가 하는 것이 그런 예겠다.

```
- setRefCount:(int)count;
- (int)refCount;
- incrementCount;
- decrementCount;
```

이런 것들은 프로토콜로 묶어 놓을 수 있겠다. 그리고 저런 프로토콜이 필요한 클래스에선 그 프로토콜에 conform 하게끔만 해 놓으면 되는 것이다.

이 기능을 잘 생각해보면, C++에서도 안될 것은 없겠다. 사실 좀 껄적지근해서 그렇지, 굳이 상속관계로 못 묶어 놓는 것은 아니지 않겠는가? 그러므로 이런 protocol 개념이 없는 C++이 충분히 시장에서 먹힐만은 하다. 하지만 순수 OOP 주의자 관점에서 보면, protocol 이란 개념은 참 잘 만든 개념같다. OOP 가 왜 나왔을까? 소프트웨어의 개발 효율 향상과, 좀더 쉬운 코딩 때문이지 않는가? 즉 코딩을 사람이 생각하는 것처럼, 객체를 만들고 각 객체들은 무슨 특성이 있고 무슨 일을 할 수 있다고 정의 해 놓고, 나중엔 그렇게 만들어진 객체를 가져다가 쓴다는게 OOP 의 기본아닌가? 그 객체가 무슨 일을 하는지는 이미 그 객체 안에 다 구현되어 있으니, 가져다 쓰는 입장에선 그

구체적인 구현에 대해선 상관할 바가 아니다.(data hiding/information encapsulation) 즉, 현실 환경을 모사한 것이 OOP 이고, protocol 은 그런 현실 환경에서 충분히 있을 법한, 하지만 C++은 따로 만들어 두지 않은, 그런 것을 구현해 놓은 것이라고 이해하면 좋겠다. 하지만 이렇게 이상적인 이유말고도 실제적인 이유에서도 protocol 은 좋아보인다. 즉 전혀 연관이 없는 클래스들이 공통적인 method 를 같는다고 할때, 한번 정의해 놓은 것을 그대로 쓸 수있게 해준다. 코드를 작성하는 입장에서 편하지 않겠는가? 모르긴 몰라도 웬지 메모리 측면에서도 좋을 것 같다. 저런 protocol pool 에 대한 포인터를 가지고 있는 것과 (내부 구현은 모르겠지만 추측이다.) 아니면 한 클래스에 대한 상속으로 인해 덩치가 커지는 것과 어떤게 더 메모리 차원에서 좋을까?

또 다른 장점으로는 객체는 class 가 아니라 protocol 로써 type 을 구분할 수도 있다는 점이다. 예를들어, NSMatrix 가 그 서브 객체인 cell 들과 뭔가 주고 받아야 한다고 하자. NSMatrix 는 각 서브 객체들이 NSCell 라는 종류이어야만 하고, 그러므로 각 서브 객체들이 NSMatrix 에서 오는 메시지에 반응하는 method 를 가진, NSCell 에서 상속을 받어야 할 것이다.
하지만 여기서 protocol 의 개념을 이용하면, 굳이 같은 NSCell 클래스에서 나온 class 가 아니더라도, 어떤 특정의 메시지에 반응하도록 만들어진 method 를 conform 하도록만, 서브 객체인 cell 들이 구성되어 있다면 상당히 유연한 코딩이 가능해진다. (뭔가 C++의 template 역할을 할 수 있다는 생각이 또 든다.)

6.2.3 Protocol 의 종류

이제까지 protocol 의 개념과 장점, 그리고 어디에 쓸 수 있는지에 대해서 알아보았다. 이번엔 protocol 의 종류에 대해서 알아보자. 우와.. 이렇게 복잡한데 거기에 또 종류가 있다고? 처음에 Objective-C 를 접할땐 이런 개념이 무척이나 복잡하게 보일 수도 있다. 먼저 Objective-C 를 접하고 C++을 접한 사람이 흡사 C++을 복잡하다고 여기듯이, 그 반대로 대부분의 사람들은 아무래도 먼저 C++을 접할테니 반대로 Objective-C 가 복잡하게 느껴질 수도 있다. 하지만, 걱정하지 마시기 바란다. zero base 로 둘다 모른다고 하자. 그러면 어떤게 더 편한 개념을 제공할까? 필자가 보기엔 Objective-C 이다. 또한 뭐든지 처음볼때는 아직 머리속에 확실히 정리가 안되었기 때문에 복잡해 보이게 마련이다. 그러므로 끝까지 읽고, 그다음에 Mac 에서 게임을 하면서 좀 쉬던가 아니면 바깥 바람이라도 좀 맞고 오던가, 잠을 한 숨 잔 후에 다시 보면 모든게 정리가 잘 되면서 쉽게 느껴질 것이다. 그러므로 걱정하지 말자.

그럼 이제 Protocol 의 종류에 대해서 알아 보자.

6.2.3.1 Informal Protocol

Category 선언내에 method 를 선언하면 그게 바로 informal protocol 이 된다.

```
@interface NSObject ( RefCounting )
- (int)refCount;
- incrementCount;
- decrementCount;
```

```
@end
```

위의 예에서 볼 수 있듯이, informal protocol 은 대개 NSObject 클래스에 대한 카테고리로써 선언한다. 그렇게 하면, NSObject 에서 나온 새끼 클래스들이 모두 다 사용할 수 있기 때문이다. 물론 다른 클래스에 대한 category 로써도 정의할 수 있다. 그럼 그 클래스에서 나온 클래스들만 사용할 수 있다.

여기서 주의할 점은, 카테고리 interface 를, protocol 을 선언하기 위해서 쓸때는, 거기에 상응하는 implementation 을 가지지 않는다는 점이다. 대신에 그 method 들을 가지는 protocol 을 구현하는 클래스들이 그 자체의 interface 파일에서 다른 method 들과 마찬가지로 구현하게 된다.

다시 말해보자. 생긴 걸로 봤을때, informal protocol 은 Category 와 완전히 같다. 단지 차이점은 Category 는 구현에 대한 부분도 역시 카테고리로 만들어서 넣어주나, informal protocol 일 경우엔 그 구현을 해당 클래스에서 method 로 직접 해 준다는 것이다.

또한 informal protocol 은 category 를 선언하는 방식을 좀 바꿔서 method 들의 리스트를 써 놓기는 하지만 어떤 특정 클래스나 구현에 고정시켜 놓지는 않는다.

informal protocol 은 구현하기가 이렇게 편하다. 하지만 단점이 있게 마련이다. 편한거치고 단점없는게 어디 있겠는가? 즉 language 로부터 그다지 많은 지원을 받지 못한다. 표로 정리해 보자.

compile time 시	type checking 을 하지 않는다
run time 시	한 객체가 그 protocol 을 conform(준수)하는지 알수 없다.

<Table 1> informal protocol 의 단점

이런 단점이 싫다면 그때는 다음에서 알아볼 formal protocol 을 사용하면 된다. 그럼 이런 informal protocol 은 언제 쓰면 좋을까? 그건, protocol 에 정의된 method 들을 구현하는게 굳이 필요하지 않을때이다. 즉 delegate 같은 것을 구현할 때 쓰면 편하다. 즉 다른 객체 대신에 일을 수행하는 객체를 만들때 좋다.

NOTE : 엄밀하게 보면 informal protocol 은 없는 것으로 보인다. 이것은 Category 의 다른 이용일 뿐이다. Category 와의 차이점은 앞에서도 나와 있지만, 그 구현을 클래스내에 하느냐 아니면 따로 준비된 곳에 하느냐의 차이이다. 필자가 Objective-C 를 처음 접하면서, 처음엔 참 쉽게 잘 만들어졌다고 생각하면서도 한동안 보지 않다가 다시 보면 헷갈리고, 복잡하다고 여긴 부분이 바로 이 부분때문이다. syntax 로 category 와 informal protocol 이 명확하게 구별이 안된다.
또한 그 사용 목적도 그렇다. Category 의 원목적이 이미 있는 클래스에 뭔가를 더 추가해 넣을때, 그리고 attribute/property 가 아닌 behaviour 로 클래스들을 구분할 때 쓰기 위함인데, 결국 protocol 과 비슷하지 않은가? protocol 도 그 protocol 을 conform 하는 것끼리의 구별하는 목적이 있다. Objective-C 는 NeXT 의 몰락과 함께 사실 그 명맥이 다했으며 C++처럼 활발한 개발과 확장, 혹은 정제가 이루어지지 않았다. 언어의 간결함에 대해선 C++보단 더 좋으나, 뭔가 아직 미완성인 듯한 느낌이 나는 부분이 바로 이 부분이다. 앞으로 Apple 이 다시 이 Objective-C 를 발전시켜 나가면 좋겠다.

6.2.3.2 Formal Protocol

이 formal protocol 은 컴파일러와 runtime 시스템이 지원을 한다. 즉 앞에서 살펴본 informal protocol 의 단점을 가지고 있지 않다는 말이다.

formal protocol 은 다음과 같이 @protocol 이라는 directive 를 이용해서 선언한다.

```
@protocol ProtocolName
method declarations
@end
```

앞에서 나온 reference-counting protocol 을 이번엔 formal 하게 해보자.

```
@protocol ReferenceCounting
- (int)refCount;
- incrementCount;
- decrementCount;
@end
```

단 protocol 이름은 global 하게 access 할 수가 없다. 즉 global visibility 가 없다. 이것은 Class 의 이름과는 사뭇 다른 점이다. protocol 은 자체의 namespace 안에만 있게 된다. 이게 무슨 소리인가? C++에서 class 를 정의한다고 하자. 그것을 myClass.h 라는 헤더 파일에 넣었다고 하고, 여러분의 소스코드중 일부는 그 헤더 파일을 include 하고 어떤 것은 include 하지 않았다고 하자.
include 한 cpp 파일에선 그 클래스 이름을 이용해서 변수도 선언하고 여러가지를 할 수 있다. include 하지 않은 파일에서도 다른 소스 파일에 선언된 객체가 있다면 extern 등을 이용해서 가져다 쓸 수 있다. 하지만 protocol 은 이게 안된다는 것이다.
그 이유는 protocol 은 클래스랑 연관을 시켜서 써주는 것이지 단독적으로 쓰는것이 아니기 때문이다. 즉 어떤 protocol 을 쓰려면 class 가 그 프로토콜을 "준수"해야 하는 것이다. 즉 그 프로토콜을 adopt 혹은 채용해야 한다. 그러면 그제서야 그 class 가 해당 프로토콜을 준수, 혹은 conform 을 하는 것이다. 어떻게 하면 되는지 예를 살펴보자.

```
@interface ClassName : ItsSuperclass < protocol list >
```

카테고리가 protocol 을 받으려면

```
@interface ClassName ( CategoryName ) < protocol list >
```

<protocol list>에 여러개의 protocol 이름을 넣고 싶으면 쉼표를 이용해서 넣으면 된다.

```
@interface ClassName ( CategoryName ) < protocol_1, protocol_2 >
```

단 protocol 을 받아들일 클래스나 카테고리 파일에서는, 반드시 그 프로토콜이 선언된 헤더파일을 import 해야만 한다. 그리고 그렇게 받아들여진 프로토콜안에 정의된 method 들은 클래스나 카테고리 인터페이스의 다른 부분에서 또 선언되면 안된다.

물론 프로토콜 method 외엔 아무 method 도 가지지 않는 클래스도 있을 수 있다.

```
@interface Formatter : NSObject < Formatting, Prettifying >
@end
```

한 클래스나 카테고리가 일단 어떤 프로토콜을 준수하겠다고 한다면,그 클래스나 카테고리는 준수하고자 하는 프로토콜에 선언된 모든 method 를 다 구현해야 한다. 그렇지 않으면 컴파일러가 경고를 하게 된다. 이건 흡사 클래스를 선언하는 것과 비슷하다고 생각하면 된다. 둘다 정의하는 method 들을 다 implemenation 해줘야 하니까 말이다.

이렇게 하여 informal protocol 과 formal protocol 에 대해서 알아보았다. 여러분들은 뭔가를 느끼지 않는가? 잠시 쉬었다가 다시 Category 를 보시기 바란다. 그리고 informal protocol 의 마지막 부분에 NOTE 라고 써 놓은 부분을 보시기 바란다. formal protocol 은 선언시에 @protocol 이란 특별한 directive 를 써 놓았을 뿐, 그 구현은 역시 informal protocol 과 마찬가지로 class 내에서 한다. Category 는 어떤가? 그 선언과 구현이 별도로 되어 있다. 즉 클래스에 종속되어 있지 않다. 그렇다면 Category 와 Protocol 의 궁극적인 차이는 무엇일까?
Objective-C 의 장점이 알지 못하는 객체에다가 메시지를 보내고, 그 헤더파일이 없는 클래스를 상속이 아닌 수평적인 수준에서 새 함수를 더 넣을 수 있다는 것이다. Category 는 이미 있는 클래스를 수평적으로 확장을 할 때, 유용하다. 단 그때, 그 클래스의 소스코드가 없을 수 있다. 그러므로 별도의 장소에서 다음과 같이 한다.

```
@implementation ClassName ( CategoryName )
        method definitions
@end
```

하지만 protocol 인 경우엔 저 “ClassName” class 자체에서 하게 된다. 즉 protocol 인 경우엔, 내가 지금 만드는 클래스가 어떤 것을 다 가져야 할지 이미 알고 있는 경우란 소리다.

Category 와 Protocol 개념은 클래스들을 그 행위로서 구분지어줄 수 있는 것들이지만, 이렇듯 미리 무엇을 할지 알고 있느냐 없느냐에 따라 용례가 갈린다고 볼 수 있다. 또한 Category 의 경우에는 뭔가 해당 클래스의 versioning 이라는 점이다. 결국 보면 protocol 과 비슷해질 수는 있겠지만, 개념적으론 좀 차이가 있다.

6.2.4 Protocol Object

여기서는 runtime 시에 protocol 이 어떻게 표현되는지에 대해서 알아보자. 클래스는 class object 로, 함수는 selector 로 runtime 시에 나타나듯이, formal protocol 은 Protocol 클래스로 나타난다. protocol 을 다루는 코드는 Protocol 오브젝트를 참조해야만 한다.

어떻게 하는지 알아보자.
protocol 객체를 참조하는 방법은 @protocol() 디렉티브를 이용해서 한다. 즉 protocol 을 선언할 때 사용했던 바로 그 디렉티브랑 같다. 따라오는 괄호만 빼면 말이다. 괄호 안에는 protocol 의 이름을 써 넣는다.

```
Protocol *counter = @protocol(ReferenceCounting);
```

class 이름과 다른 점은, protocol 이름은 @protoco()안에 있는 것을 제외하고는, 어떤 한 객체를 의미하지는 않는다.

컴파일러는 다음의 경우에만 protocol 객체를 만든다.

- 클래스에 의해서 adopt 되었을 때,
- 소스코드에서 다른 어딘가를 참조했을 때 (@protoco()을 사용해서..)

즉 선언은 되었지만 사용되지 않는 protocol 은 runtime 시에 protocol 객체로 나타나지 않는다.

6.2.5 프로토콜을 준수한다는 것

프로토콜을 준수한다는 것, 즉 conform 한다는 것은, 한 클래스나 객체가 그 프로토콜의 모든 method 를 다 구현하고 있다는 것이다.
어떤 클래스가 주어진 protocol 을 준수하는지 알아보려면 conformsToProtocol:이라는 메시지를 해당 객체에 보내면 된다.

```
if ( [receiver conformsToProtocol:@protocol(ReferenceCounting)] )
[receiver incrementCount];
```

이 confromsToProtocol: 메시지는 앞에서 살펴본 respondsTo:와 매우 흡사하다.(protocol 에 대한 설명의 처음 부분에 나온 예에 나와 있다.) 차이점이 무엇이겠는가? conformsToProtocol:은 해당 객체가 주어진 프로토콜을 준수하는지, 즉 그 프로토콜에 정의된 모든 method 를 다 가지고 있는지를 알려준다면, respondsTo 는 주어진 특정 method 를 가지고 있는지만을 알려준다. 즉 respondsTo 는 conformsToProtocol 의 부분이라고 보면 되겠다.
또한 어떤 면에서는 isKindOfClass:와도 비슷하다. isKindOfClass 는 상속에 의한 소속을 알려준다.

	conformsToProtocol:	respondsTo:	isKindOfClass:
목적	receiver 가 주어진 프로토콜을 준수하는지 여부를 알려준다.	receiver 가 주어진 메시지에 반응하는지 알려준다. 즉 그 메시지를 receiver 가 구현하는지를 알려준다.	receiver 가 주어진 class 종류인지 알려준다.

용례	[receiver conformsToProtocol:@protocol(protocol_name)]	[anObject respondsToSelector:@selector(function_name::)]	

6.2.6 type checking

타입을 검사하는 것은 formal protocol 에 대해서까지 할 수 있다. 일반적인 타입 검사에 비해 하나의 장점은 protocol 까지 체크를 하면 좀더 abstract 한 부분까지 가능하게 된다는 점이다. 예를 들어 보자.

```
- (id <Formatting>)formattingService;
id <ReferenceCounting, AutoFreeing> anObject;
```

static typing 처럼 위의 것도 클래스의 계층 구조에 따라 테스트를 하지만, 또한 protocol 을 준수하는지 여부에 따라서도 검사를 한다. 예를들어 다음의 선언을 보자.

```
Formatter *anObject;
```

이렇게 선언을 하면, Formatter 의 child class 들을 모두 cover 할 수 있으며, 다 Formatter 클래스인지 여부를 체크할 수 있다. 마찬가지로 다음은 프로토콜에 대해서 비슷한 것을 하게 해 준다.

```
id <Formatting> anObject;
```

이렇게 하면 위의 anObject 는 Formatting 을 conform 하는 모든 객체를 다 가르킬 수 있게 된다. 이때 클래스의 계층 구조와는 상관이 없게 된다. 그러므로 Formatting protocol 을 준수하지 않는 객체를 anObject 로 가르키려 한다면, 컴파일러가 에러메시지를 내게 된다.

물론 위의 두가지 형태를 합칠 수도 있다.

```
Formatter <Formatting> *anObject;
```

단 protocol 은 class object 의 타입에 대해서는 쓸 수없다. instance 들 만이 protocol 에 정적으로 타이핑되기 때문이다. 하지만 runtime 시에는 클래스와 instance 들이 다 conformsToProtocol: 메시지에 반응하기는 한다.

6.2.7 protocol 안에서 또 protocol 쓰기

한 protocol 은, 흡사 클래스가 protocol 을 채용할 때 썼던 그런 문법으로, 다른 protocol 을 포함할 수 있다. 이것을 nested protocol 이라고 불러야할지는 모르겠다. 그런 용어가 있는지도 모르겠으며, 엄밀히는 nested 는 아니기 때문이다.

예를 보자.

```
@protocol ProtocolName < protocol list >
```

이렇게 하면 ProtoclName 이라는 protocol 은 protocol list 에 준 모든 프로토콜을 다 포함하게 된다. 즉,

```
@protocol Paging < Formatting >
```

이렇게 하면 Paging 이란 protocol 은 Formatting 이란 protocol 도 포함하게 되는 것이다. 그러므로,

```
id <Paging> someObject;

if ( [anotherObject conformsToProtocol:@protocol(Paging)] )
```

들은 Paging protocol 에 대한 conformance 테스트만 하면 Formatting 에 대한 것은 자동으로 되는 것이다. 이렇게 여러개의 protocol 을 conform 하는 class 를 만들려면 다음과 같이 하면 된다.

- conform 하는 모든 protocol 의 method 를 다 구현하던가
- 해당 protocol 을 수용하고 구현해 놓은 클래스를 상속받던가

예를 들어

```
@interface Pager : NSObject < Paging >
```

라고 되어 있다면, Pager 는 Paging 프로토콜을 다 구현해야 함은 물론이고 Formatting 프로토콜도 다 구현해야 한다.

하지만 다음 예를 보자.

```
@interface Pager : Formatter < Paging >
```

이렇게 하면 이미, Formatting 프로토콜은 이미 Formatter 에서 구현되어 있고, 그것을 그대로 상속 받았기 때문에 Pager 에서는 Paging 프로토콜만 구현하면 된다.

NOTE : Class 가 protocol 을 formal 하게 채용하지 않아도, method 로써 protocol method 들을 구현하면 자동으로 그 protocol 을 준수하게 된다.

6.2.8 다른 protocol 을 언급하기

여기서는 protocol 을 사용하기 전에 이런 protocol 을 쓰겠다고 미리 언급해두는 것에 대해서 알아 보겠습니다. 역시 Apple 의 문서에서 코드를 가져와서 보자면..

File A	File B
#import "B.h" @protocol A – foo:(id <B>)anObject; @end	#import "A.h" @protocol B – bar: (id <A>)anObject; @end

가만히 두 파일을 놓고 보면, 둘 사이에 cyclic dependency 가 있다는 것을 알 수 있다. 쉽게 풀어서 말해보자. A 에 있는 protocol A 에서는 protocol B 를 사용하고자 한다. 반면 B 에 있는 protocol B 에서는 protocol A 를 사용하려고 한다. 그러면 A 입장에서는 B 가 컴파일 되어 있어야 B 의 프로토콜을 알테고, B 의 입장에서는 A 가 컴파일되어 있어야 A 의 프로토콜을 알 수 있을것이다. 이렇게 서로간에 의존성이 있을때, 그리고 그게 연결이 되어서 cycle 을 이룰때 cyclic dependency 가 있다고 한다. 가능하면 이런 코드를 짜지 말기 바란다. 이렇게 코드를 짜면 심각한 문제가 있을 수 있다. 필자가 관여하고 있는 프로젝트엔 이런 cyclic dependency 가 심하게 많다. 그러므로 batch 빌드가 불가능하다. 일단 구버젼이라도 바이너리 오브젝트가 만들어져 있던가, 해당 library 가 구버젼일지라도 있으면 그래도 컴파일되지만, 애초에는 곤란하다. GNU 프로젝트에도 이런 식으로 된 프로젝트가 있는데, 문서에 보면 컴파일을 두어번은 하고서 쓰라고 언급되어 있다. 그래도 이 경우는 컴파일은 되는 경우인데, 왜 두어번하라고 하냐면, 한번 컴파일 시에 나오는 object 파일은 해당 소스코드의 모든 내용을 제대로 담고 있지 못하기 때문이다. 이렇듯 cyclic dependency 가 있게 되면, 여러가지 문제가 있다. 그러므로 가능하면 이런 경우를 없애는게 좋다.

이 경우 어떻게 하면 컴파일이 되게 할 수 있겠는가? 다음을 보자.

```
@protocol B;

#import "B.h"
@protocol A
- foo:(id <B>)anObject;
@end
```

A 파일의 앞에 @protocol B;를 넣었다. 즉 B 라는 것은 어디선가 정의된 protocol 이름이다라고 명시해 두는 것이다. 이것은@class 와 같은 내용이다. Class 인 경우에서도 @class 라는 것을 이용해서 이렇게 미리 명시했었다. 일종의 C 나 C++에서의 extern 이 하는 역할 중 하나와 비슷하다고 보면 되겠다.

이렇게 바꾸면 문제없이 컴파일이 된다. 하지만 명심하자. 아주 아주 꼭 필요한 경우가 아닌다음에는 이런 상황이 가능하면 발생하지 않게 하자. cyclic dependency 가 아닌 경우라도 이렇게 명시를 해야 할 필요가 있을 것이다. 그럴때는 아주 유용하게 쓸 수있겠다.

Chapter 7. Enabling Static Behaviors

이 장에서는 Objective-C 에서 static typing 등 정적인 요소에 대해서 알아보겠습니다. 지금까지 알아본 messaging, protocol 그리고 category 등은 언어자체에 상당히 동적인 요소를 추가해 주는 기능이었습니다. 바로 이런 동적인 측면이 Objective-C 의 큰 특징이자 SmallTalk school 쪽 언어들의 큰 특징입니다. OOP 는 동적인 측면과 뗄래야 뗄 수가 없으므로 Objective-C 언어에 대한 책이나 문서들은 이런 동적인 측면을 많이 강조합니다. 혹자들은 Objective-C 가 동적이기에 유연하지만 느리다라고도 합니다. 맞는 주장입니다. 하지만 그렇다고 해서 Objective-C 가 정적인 면을 못쓰는게 아닙니다. Interface Builder 에서 GUI 코드를 생성하게 되면 default 로 id 타입으로 class 들이 선언됩니다만 이것들을 static typing 으로 바꿀 수 있습니다. 이 static typing 을 많이 사용하면 속도 측면에서, 그리고 compile 을 할때 보다 더 정확한 에러 메시지를 볼 수가 있는 등 장점이 많습니다. 그러므로 Objective-C 가 동적인 언어라고해서 동적인 면만 쓸 것이 아니라 정적인 것과 적절한 조화를 이루어 쓰면서, 유연함과 성능의 두 토끼를 잡는 것이 좋을 것입니다.

자, 들어가기에 앞서서 Objective-C 의 동적인 측면을 정리해 봅시다.

- 객체를 위해 메모리가 동적으로 할당 됩니다.
- 객체들은 동적으로 type 이 결정 됩니다. 컴파일 시에는 object pointer 가 id 이어서 어떤 객체이건 가르킬 수 있지만, 그 실제적인 type 은 runtime 시에 결정됩니다.
- 메시지들은 동적으로 바인딩됩니다. selector 를 이용해서 Objective-C runtime 시스템이 메시지를 receiver 들과 동적으로 바인딩 시킵니다. 이것은 C++의 virtual 을 이용한 late binding 과 같다고 생각하면 쉽게 이해가 될 것입니다.

그러면 이제 static typing 을 하게 되면 얻는 장점을 살펴 봅시다.

- compile 시에 type checking 이 됩니다. 이것은 에러를 더 쉽게 찾을 수 있게 합니다.
- 코드가 self-documenting 이 되게 합니다. 동적인 것은 코드만 봐서는 실제 runtime 시에 어떤 것이 call 될지 어떤 타입이 될지 알 수 없습니다. 하지만 정적으로 만들면 굳이 코드를 실행해 보지 않고도 파악할 수 있습니다.
- runtime 에 하는 양을 줄여 compile time 으로 옮길 수 있습니다. 그만큼 runtime 시에 성능이 좋아집니다. 하지만 Objective-C 의 runtime 효율이 좋기 때문에, 아주 필요한 경우가 아니라면 동적인 면이 많더라도 그리 성능이 나쁘진 않습니다.

7.1 Static Typing

C 와 C++에서의 선언과 같이 해주면 됩니다. 즉 id 대신에 실제 사용할 데이터 타입을 이용해서 선언합니다.

```
Rectangle *thisObject;
```

이렇게 하면 thisObject 는 Rectangle 타입이나 Rectangle 로 부터 상속받은 클래스만 pointing 할 수 있습니다.

NOTE : id 는 Objective-C 만의 전유물이 아닙니다. C/C++에서도 void pointer 를 이용해서 할 수 있습니다. 하지만 최근의 C++은 예외가 되겠지만, 기존에는 C++에서는 수행중에 그 객체의 타입을 알 수가 없었습니다. 그러므로 void pointer 를 이용하는 경우는 특별한 경우로 제한 되어 있었습니다. Objective-C 에선 특히 Cocoa 와 연결이 되면 대부분의 객체는 NSObject 를 super class 로 합니다. 그런데 이 NSObject 에는 runtime 시에 어떤 타입이다라는 것을 기록해 놓게 되어 있습니다. 그러므로 id 를 이용해도 언어가 동적인 상태에서 주어진 임의의 포인터인 id 를 제대로 해석할 수 있는 것입니다. 현재의 C++은 RTTI 가 지원된다는 사실을 염두에 둡시다.

단 이렇게 하면 typing 을 정적으로 한다는 뜻이지 allocation 도 꼭 정적으로 한다는 뜻은 아닙니다.

```
Rectangle *thisObject = [[Square alloc] init];
```

id 를 쓸때와 마찬가지로 위의 코드는 thisObject 를 동적으로 할당합니다.
또한 아주 정확한 typing 은 비록 정적으로 되어 있다고 해도, runtime 시에 결정이 됩니다. thisObject 가 꼭 Rectangle 타입만이 아니라 그 밑의 child class 들도 pointing 할 수 있으므로, 실질적인 typing 은 runtime 시에 결정이 됩니다. 즉 위의 예에서

```
[thisObject display];
```

라고 하면 Square 의 display 를 호출합니다. C++이라면 Rectangle 클래스에 display method 가 있다면, Rectangle 의 것을 호출했을 겁니다. Objective-C 와 C++의 차이점은 Objective-C 는 late binding 이 우선한다는 것입니다. C++에서는 display 가 virtual 로 선언되어 있었다면 Square 의 것이 호출될 것입니다.

정적으로 typing 을 하게 되면 프로그래머로써 실질적으로 도움이 되는 다음과 같은 장점이 있습니다.

- compile 시에 타입을 검사합니다.
- 같은 이름을 가진 method 는 같은 return 과 argument/parameter 타입을 가져야 한다는 제한에서 벗어날 수 있습니다.
- structure pointer 를 이용해서 바로 객체의 instance variable 들을 억세스 할 수 있습니다. 굳이 get / put 모델을 사용하지 않아도 된다는 뜻입니다.

NOTE : get / put 모델은 여러명이 참여하는 대규모 프로젝트에서 정말 유용합니다. 사람들마다 다 다른 방식으로, 즉 A 라는 사람이라면 그런 method 이름은 값을 세팅하는데 써야지라고 생각할 만한

건데 다른 사람은 값을 받아 오는데 쓸 수도 있고, 전혀 다른 용도로 쓸 수도 있을겁니다. 그러므로 get / put 모델은 OOP 의 프로그래밍 모델에도 부합할 뿐 아니라, 쉽게 이해할 수 있는 코드를 만드는데도 도움이 됩니다. 요새는 MS 조차 ActiveX 나 COM 프로그래밍에서 get / put 모델을 사용한다는 점을 생각해 봅시다.

하지만 instance variable 을 바로 써도 문제는 없겠습니다. 오히려 public variable 같은 경우엔 함수를 call 하지 않아도 되므로 성능이 더 좋아질 겁니다.

여러가지 코딩 스타일 중 그때 그때 제일 적합한 것을 골라 쓰는 것이, "어느 한 모델이 좋아"라고 하는 것보단 더 합리적일 겁니다.

7.1.1 Type Checking

정적으로 typing 을 하면 compiler 가 다음의 두가지 방식으로 더 좋은 효율을 내게 해 줄 수 있습니다.

- 메시지가 static 하게 type 이 결정되있는 receiver 에 전해졌을때, compiler 는 receiver 가 그 메시지를 처리할 수 있는지 알 수 있습니다. 만약 처리하지 못한다면 compile 시에 warning 을 내어 줍니다.
- static 하게 type 이 결정되어 있는 객체가 역시 static 하게 type 이 결정되어져 있는 변수에 할당될 때, 컴파일러는 두 type 이 호환성이 있는지 파악할 수 있습니다. 만약 그렇지 않다면 warning 을 내어 줍니다.

두번째의 경우에서 호환성이 있으면 warning 이 나지 않는데 그 예를 봅시다.

```
Shape *aShape;
Rectangle *aRect;
aRect = [[Rectangle alloc] init];
aShape = aRect;
```

Rectangle 의 super class 가 Shape 일때, 아무런 warning 을 내어주지 않습니다. 하지만 만약

```
aRect = aShape;
```

으로 반대로 되어 있으면, (물론 이때는 aShape 쪽이 메모리가 할당되어 있습니다.) warning 을 냅니다. 그 이유는 모든 Shape 형이 Rectangle 형이 아니기 때문입니다.

만약 양쪽 둘 중의 하나만이라도 타입이 id 라면, 그때는 type checking 을 하지 않습니다.

7.1.2 Return 과 Argument 타입

일반적으로 서로 다른 클래스에 정의된 method 들은 같은 selector, 즉 같은 이름을 가질 수 있습니다. 이것은 polymorphism 에 의거합니다. 하지만 동적으로 타이핑 된 경우는, 같은 타입의 return 과 argument 를 가져야 합니다. 그 이유는 메시지를 받는 receiver 의 class 를 compile 시에 알수가 없기에, compiler 는 이름이 같은 method 들은 어떤 클래스에 있는 것이건 다 같은 식으로 처리하기 때문입니다.

정적으로 타이핑이 된 객체에 메시지가 보내질때는, receiver 의 클래스를 compiler 가 알수 있으므로, method 들이 다른 argument 와 return 타입을 가질 수 있습니다.

7.1.3 상속된 클래스에 대한 Static Typing

instance 는 자신의 클래스에, 혹은 상속받은 클래스, 즉 super class 에 static 하게 typing 될 수 있습니다. 그러므로 궁극적으로는 모든 인스턴스들이 NSObjects 로써 static 하게 타이핑될 수 있습니다.

하지만 컴파일러는 클래스 이름을 이용해서 그 타입이 뭔지를 파악하기 때문에, 한 instance 를 그 super class 의 타입으로 하면, 컴파일러가 runtime 시에 어떻게 될거라고 생각하는 것과 실제의 type 과는 달라질 수가 있다는 점을 주의하십시오.

무슨 말인지 어렵습니다. 쉽게 이해하기 위해, 예를 봅시다.

```
Shape *myRect = [[Rectangle alloc] init];
```

컴파일러는 myRect 를 Shape 로 취급하게 됩니다. 그런데 Rectangle 의 method 를 수행하려고 메시지를 다음과 같이 보낸다고 합시다.

```
BOOL solid = [myRect isFilled];
```

그럼 컴파일러는 경고를 내게 됩니다. isFilled 는 Shape 에는 정의되어 있지 않기 때문입니다. 자 그럼 이제는 Shape 에 정의된 것을 호출한다고 합시다. 또한 Rectangle 도 그 method 를 정의하고 있다고 합시다.

```
[myRect display];
```

이때는 compiler 가 아무런 warning 도 내지 않습니다. 비록 Rectangle 이 display 라는 method 를 가지고 있다고 하더라도 myRect 가 Shape 로 static 하게 타이핑 되어 있으므로 Shape 의 display 를 호출합니다.

7.2 Method 의 주소를 얻기

동적 바인딩을 하지 않는 방법으로는 호출할 method 의 주소를 얻어와서 그것을 강제로 부르는 방법이 있습니다. 주소를 얻어오려면 NSObject 클래스에 정의된 *methodForSelctor:*라는 method 를 호출하면 됩니다. 그리고 받아온 주소를 가지고 해당 프로시져를 호출하면 됩니다. 이때, return 된 원래 의도했던 그 펑션의 것으로 cast 가 되어야만 합니다. return 과 argument 타입이 다 cast 할때 사용되어야 합니다.

```
void (*setter)(id, SEL, BOOL);
int i;
setter = (void (*)(id, SEL, BOOL))[target methodForSelector:@selector(setFilled:)];
for ( i = 0; i < 1000, i++ )
        setter(targetList[i], @selector(setFilled:), YES);
```

setter 로써 호출되는 함수에서 첫번재 인자는 method 를 받는 객체(self)이고, 두번째는 selector (_cmd)입니다. 이 둘은 method syntax 에서는 안보이지만, function 으로 명시해서 호출할때는 이처럼 꼭 써 주어야 합니다.

methodForSelector:를 이용해서 동적 바인딩을 하지 않는 것은, 메시징처리를 위해 필요한 많은 시간을 절약할 수 있습니다. 하지만, 위처럼 loop 을 이용해서 특정 메시지가 계속처리되는 경우정도에만 그 성능 향상이 현저히 날 것 입니다.

NOTE : methodForSelector:는 Cococa runtime 이 제공하는 것이지, Objective-C runtime 이 제공하는 것은 아닙니다.

7.3 객체의 data structure 를 얻기

OOP 개념에서는 data hiding 이 기본 개념입니다. 즉 한 객체의 내부에 있는 정보를 알아내려면, 그 객체에 적절한 메시지를 보내서, 그에 대한 답으로써 얻어내는 것이 기본이지, 그 내부를 직접 드러내지는 않습니다. 하지만 Objective-C 는 한 객체의 내부 정보를 흡사 C 의 structure 처럼 드러나게 하는 기능이 있습니다. 하지만 이런 식으로 프로그래밍하는 것을 권장하지는 않습나다. 꼭 필요한 경우만 하십시오.

@def()라는 directive 에 클래스의 이름을 인자로써 전달해 주면, 해당 클래스의 인스턴스에 대한 declaration list 를 얻을 수 있습니다. 이 리스트는 구조체를 선언할때만 씁니다. 그러므로 @def()는 구조체의 선언문 내부에서만 쓸 수 있습니다. 다음의 코드는 Worker 클래스의 구조를 그대로 반영하는 구조체를 선언합니다.

```
struct workerDef {
        @defs(Worker)
} *public;
```

이 예에서 public 은 포인터로써, Worker 인스턴스와 동일한 구조를 갖는 구조체를 가르키기 위해서 사용되겠습니다. Worker 의 id 는 이 pointer 에 할당될 수 있습니다. 그리고 난 후에, 그 객체의 인스턴스 변수들은, 그 public 이라는 포인터를 이용함으로써 억세스할 수 있습니다.

```
id aWorker;
aWorker = [[Worker alloc] init];
public = (struct workerDef *)aWorker;
public->boss = nil;
```

이렇게 하면, 주어진 객체안의 변수들이 비록 @private 이나 @protected 로 선언되어 있다 할지라도 모두 다 public 처럼 억세스될 수 있습니다.

이렇게 하는 방법은 프레임워크 내의 클래스나 혹은 여러분이 직접 만들었지만 프레임워크에 있는 클래스로부터 상속한 클래스를 위해서는 쓰지 마십시오.

Chapter 8. Exception Handling 과 Thread Synchronization

Objective-C 는 exception handling 과 thread synchronization 을 지원합니다. 이 기능들을 사용하려면 compile 할 때, -fobjc-exceptions 스위치를 키십시오. 즉 GCC 의 인자로써 넣고 실행하면 됩니다. 이것은 gcc 3.3 이상의 버젼에서 사용가능합니다.

NOTE : 이 기능 중 하나라도 사용을 하게 되면, 그렇게 해서 만들어지는 프로그램은 MacOS X 10.3 버젼과 그 이후의 버젼에서만 사용할 수 있습니다. 이 기능들은 그 이전의 OS 에선 지원되지 않습니다.

이 말은 Objective-C 가 자체적으로 지원하는 기능에 대해서 그렇다는 것입니다. Cocoa 에서 NSException 이나 NSThread 같은 것이 있으므로, 또한 대개 MacOS X 프로그램은 그것들을 이용해서 만들어질 것이므로, 이런 기능이 지원되지 않는다고는 생각하지 마시기 바랍니다. 또한 Unix 에서 많이 쓰는 pthread 같은 외부 라이브러리로 지원되는 thread 도 있으므로, 기능에 제약이 따른다거나 그럴 것은 없습니다.

8.1 Exception 처리하기

Objective-C 는 Java 나 C++과 유사한 exception handling 문법을 가지고 있습니다. 하지만 NSException 이나 NSError 와 같은 클래스를 이용해서도 에러를 다루는 코드를 넣을 수 있습니다.

exception 을 지원하는 것은 네 compiler directive 를 이용해서 하게 됩니다. 그것들은 다음과 같은 것이 있습니다.

@try
@catch
@throw
@finally

exception 을 던질만한 코드는 @try 블럭으로 감싸십시오. 그리고 @catch 블럭은 @try 블럭에서 발생하는 혹은 던져지는 exception, 즉 에러를 처리하는 코드가 들어가게 됩니다. 그리고 @finally 블럭은 exception 이 던져지건 안던져지건 최종적으로 수행되어야할 코드가 들어가게 됩니다. exception 을 던지려면 @throw directive 를 쓰십시오. 이때 던져지는 exception 은 기본적으로 Objective-C 의 객체에 대한 포인터입니다. NSException 객체에 대한 포인터도 사용할 수가 있지만 다른 것을 사용해도 됩니다.
자 예를 봅시다.

```
Cup *cup = [[Cup alloc] init];
```

```
@try {
        [cup fill];
}
@catch (NSException *exception) {
        NSLog(@"main: Caught %@: %@", [exception name], [exception reason]);
}
@finally {
        [cup release];
}
```

8.1.1 Exception 던지기

exception 을 던지려면, 우선 exception 에 대한 적절한 정보를 가진 객체를 만들어야 합니다. 그 정보는 exception 이름과 왜 던져졌는가에 대한 것등이 되겠습니다. 예를 보겠습니다.

```
NSException *exception = [NSException exceptionWithName:@"HotTeaException"
                                        reason:@"The tea is too hot" userInfo:nil];
@throw exception;
```

또한 @catch 블럭 안에서, 잡은 exception 을 다시 던질 수 있습니다. 이때 @throw directive 를 쓰는데, 단 argument 는 쓰지 않습니다.

이 NSException 이 여러분의 목적에 맞지 않는다면, NSException 을 sub-classing 해서 새로운 클래스를 만들어 쓸 수도 있습니다.

NOTE : 꼭 NSException 만 던질수 있는 것은 아닙니다. 실제론 어떤 Objective-C 객체든지 던질 수 있습니다. NSException 은 단지 exception 을 처리하는 루틴을 제공해서 편하게 해줄 뿐입니다. 여러분이 각자의 목적에 맞는 것을 구현해서 쓸 수가 있습니다.

8.1.2 Exception 처리하기

@try 블럭에서 던져진 exception 을 잡으려면, 한개나 그 이상의 @catch 블럭을 @try 블럭 뒤에서 써서 잡을 수 있습니다. @catch 블럭은 가장 구체적인 것에서부터 가장 포괄적인 것 순으로 작성해야 합니다. 만약 그 반대로 하면, 무조건 제일 포괄적인 것에 먼저 다 걸려 버리기 때문입니다.

```
@try {
        ...
}
@catch (CustomException *ce) {          // 1 : 가장 구체적인 것을 먼저 잡습니다.
        ...
}
@catch (NSException *ne) {              // 2 : 좀더 포괄적인 것을 잡습니다.
```

```
            // Perform processing necessary at this level.
            ...
            // Rethrow the exception so that it's handled at a higher level.
            @throw;                 // 3 : 잡은 exception 을 다시 던집니다.
    }
    @catch (id ue) {
            ...
    }
    @finally {                      // 4 : 최종적으로 해주어야 할 cleanup 프로세싱등을 여기에 넣을 수 있습니다.
            // Perform processing necessary whether an exception occurred or not.
            ...
    }
```

3 에서 던진 것을 다시 잡을 수도 있습니다. 이때 @try.. @catch 블럭을 nesting 해서 쓰면 됩니다.

8.2 Thread 실행을 synchronizing 하기

Objective-C 는 multithreading 을 지원합니다. 즉 두개 이상의 thread 가 한 객체를 동시에 억세스해서 변경을 할 수 있는 경우가 생긴다는 의미입니다.. 이런 경우는 프로그램에 있어서 심각한 문제가 됩니다. 코드의 어떤 부분이 두 thread 에 의해서 동시에 억세스 되는 것을 막기 위해서, Objective-C 는 @synchronized() directive 를 지원합니다.

이 @synchronized() directive 는 코드의 한 섹션을 잠가 놓아서 한 thread 만이 쓸 수 있도록 합니다. 억세스하지 못하는 다른 thread 는, 기존에 억세스하던 thread 가 일을 마치고 그 섹션에서 빠져나가면 그제서야 억세스할 수 있습니다.

@synchronized()는 Objective-C 의 객체라면 어떤 것이나 그 인자로 받아들일 수 있습니다. self 까지도 됩니다. 이렇게 사용되는 객체는 ***mutual exclusion semaphore*** 혹은 ***mutex*** 라고 부릅니다. 그리고 그렇게 보호되는 section 을 ***critical section*** 이라고 합니다. 다른 critical section 을 보호하려면, 다른 semaphore 를 사용해야합니다. race condition 을 막으려면, 프로그램이 멀티쓰레드화 되기 전에, 모든 mutex 를 다 생성해 놓는것이 좋습니다.

다음의 예는 self 를 mutex 로써 사용하는 것인데, instance 레벨에서 method 호출을 synchronize 하지 않을거라면, 이렇게 mutex 를 하나만 사용해도 괜찮다. 예를 들어 두 thread 가 같은 클래스의 두 instance 를 각각 따로 만들면, 둘다 동시에 같은 synchronize 화된 method 를 부를 수 있다. 이런 경우가 되면 synchronize 해준 의미가 없겠다. class 에 대해서도 마찬가지이다. self 가 class 자체를 가르키고 있을 때는, 한 thread 만이 method 를 수행할 수 있다.

```
    - (void)criticalMethod
    {
            @synchronized(self) {
                    // Critical code.
```

```
                ...
        }
}
```

다음의 예에선 현재의 selector 인 _cmd 를 mutex 로 사용한다. 이런 경우는 synchronize 되는 method 가 딱 하나만 존재할 때 유용하다. 왜냐하면 이럴 경우에만, 다른 객체나 클래스가 이름은 같지만 다른 method 를 실행할 가능성이 없기때문이다.

```
- (void)criticalMethod
{
        @synchronized(NSStringFromSelector(_cmd)) {
                // Critical code.
                ...
        }
}
```

다음의 예에선 보통의 방식을 보여준다. critical section 에 접근하기 전에 Account 클래스로부터 mutex 를 할당 받은 후, 그것을 critical section 을 lock 하기 위해서 사용한다. Account 클래스는 그 자체의 initialize method 에서 semaphore 를 만들 수 있다.

```
Account *account = [Account accountFromString:[accountField stringValue]];

// Get the semaphore.
id accountSemaphore = [Account semaphore];

@synchronized(accountSemaphore) {
        // Critical code.
        ...
}
```

Objective-C 의 synchronization 은 recursive 하고 reentrant 한 코드를 지원한다. thread 는 동일한 semaphore 를 recursive 한 방식으로 사용할 수 있다. 이때 다른 thread 들은 이미 사용하고 있는 thread 가 모든 일을 다 마치기 전가지는 해당 코드를 사용할 수 없다.

synchronized()안에 있는 코드가 exception 을 던지면 Objective-C 런타임이 그것을 잡고, semaphore 를 놓아준다. 그러므로 다른 thread 들이 ciritical section 을 사용할 수 있게 된다. 그리고 runtime 은 다음 exception handler 에게 발생한 exception 을 던진다.

Chapter 9. Objective-C 와 C++을 같이 사용하기

Objective-C 컴파일러는 같은 파일에서 C++과 Objective-C 코드를 섞어 쓰는 것을 허용합니다. 이렇게 섞어 쓴 코드를 Objective-C++이라고 부릅니다. 이렇게 했을때 얻을 수 있는 이득은, 기존의 C++ 라이브러리 코드를 Objective-C 프로그램에서 가져다 쓸 수 있다는 것입니다.

Objective-C++은 C++ 기능을 Objective-C 에 부여해주진 않습니다. 그 반대도 마찬가지입니다. 무슨 말인가 하면, C++ 객체를 호출하기 위해서 Objective-C 문법을 쓸 수 없다는 말입니다.
정리해 봅시다.

- Objective-C 객체에 C++ constructor 나 desctructor 를 쓸 수 없습니다.
- this 와 self 를 서로 바꿔 쓸 수 없습니다.
- 클래스 계층구조는 분리됩니다. C++ 클래스는 Objective-C 클래스에서 상속받을 수 없으며, 반대도 마찬가지입니다.
- C++에서 발생한 exception 을 Objective-C 가 catch 할 수 없으며, 그 반대도 마찬가지입니다.

다음의 섹션에서는 Objective-C++로 무엇을 할 수 있는지 설명합니다.

9.1 Objective-C 와 C++의 언어 기능을 함께 사용하기

Objective-C++에선 다음과 같은 것을 할 수있습니다.

- Objective-C 나 C++의 method 를 호출할 수 있습니다.
- 객체에 대한 pointer 들은 소스 코드내 어디서나 쓸 수 있습니다. 예를 들어 C++ 클래스의 데이터 멤버로써 Objective-C 객체의 포인터를 쓸 수 있으며, Objective-C 클래스의 인스턴스 변수로 C++객체의 포인터를 쓸 수 있습니다.

예를 보겠습니다.

```
// HelloWorld.mm
#import <Cocoa/Cocoa.h>

class HelloWorld;

@interface PLog: NSObject {
        HelloWorld *ptr;
}
- (void)sayHello;
- (void)sayHi: (HelloWorld *)p;
- (id)init;
```

```
- (void)dealloc;
@end

class HelloWorld {
        id printLog;

    public:
        HelloWorld(bool b) { if(b) printLog = [[PLog alloc] init]; }
        ~HelloWorld() { [printLog release]; }
        void sayHi() { printf(&rdquo;Hi&rdquo;); }
        void sayHello() { [printLog sayHi: this]; }
};

@implementation PLog
- (void) sayHello { NSLog(@"Hello, World!"); }
- (void) sayHi: (HelloWorld *)p { p->sayHi(); }
- (id) init { [super init]; ptr = new HelloWorld(false); return self; }
- (void) dealloc { delete ptr; [super dealloc]; }
@end
```

또한 언어에 따른 조건부 컴파일을 하기 위해서 Objective-C compiler 는 *__cplusplus* 와 *__OBJC__* preprocessor 를 제공하고 있다. 이것은 C++와 Objective-C 언어 자체의 표준과 일치한다.

앞에서 언급했듯이 Objective-C++은 Objective-C 객체에서 C++ 클래스를 상속받을 수 없고, 그 반대도 마찬가지이다. 그러므로 이 코드는 에러를 발생한다.

```
class Base { /* ... */ };
@interface ObjCClass: Base ... @end // ERROR!
class Derived: public ObjCClass ... // ERROR!
```

C++ 객체는 특별한 예외를 제외하곤 static 하게 type 이 결정되기 때문에, 객체 모델 자체가 Objective-C 와는 틀리다. 그러므로 서로 호환성이 없다. 좀더 정확하게 말하자면, Objective-C 의 객체와 C++의 객체가 메모리에 위치하는 모습이 완전히 다르다. 그래서 두개가 서로 상속을 받을 수 없는 것이다.

주의할 점이 있는데, Objective-C 인터페이스 안에 선언된 C++클래스는 global scope 를 갖는다는 것이다. 즉 해당 Objective-C 인터페이스 안에서만 쓸 수있는게 아니다. 이것은 표준 C 에서 nested structure 가 file scope 를 갖는것처럼 일관성을 갖는 것이다. 역시 예를 보자.

```
@interface Foo {
        class Bar { ... } // OK
}
```

```
@end

Bar *barPtr; // OK
```

Bar 라는 C++ 객체를 Foo 인터페이스 바깥에서 사용함에도 불구하고 에러를 내지 않는다. 즉 C++객체는 file scope 를 갖는다는 것이다. 여기서 주목할 점은 C++ 클래스가 Objective-C 인터페이스에서 "선언"되었다는 것이지 Objective-C 객체의 변수가 아니라는 점이다.

Objective-C 인터페이스 안에 또한 C 의 구조체를 인스턴스 변수로 넣을 수도 있다. 역시 이건 해당 인터페이스 바깥에서 선언되었건 아니건 상관 없다.

```
@interface Foo {
        struct CStruct { ... };
        struct CStruct bigIvar; // OK
} ... @end
```

C++의 클래스를 Objective-C 의 인스턴스 변수로 쓰는 것도 가능하지만, 단 virtual member function 이 선언된 C++ 클래스는 안된다.

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

struct Class0 { void foo(); };
struct Class1 { virtual void foo(); };
struct Class2 { Class2(int i, int j); };

@interface Foo: NSObject {
        Class0 class0; // OK
        Class1 class1; // ERROR!
        Class1 *ptr; // OK—call 'ptr = new Class1()' from Foo' init,
        // 'delete ptr' from Foo's dealloc
        Class2 class2; // WARNING - constructor not called!
        ...
@end
```

하지만 위의 예에서 알수 있듯이 포인터를 쓰면 어느 정도 가능하다는 점을 팁으로 알고 있으면 좋겠다. 또한 constructor 가 호출되지 않는 경우도 주목하자. 왜 이런가하면, C++ 클래스가 virtual 함수를 가지고 있으면 virtual function table 을 가지게 되는데 Objective-C 의 런타임이 이것을 이해하지 못한다. 또한 C++의 constructor 나 destructor 도 처리하지 못한다.

Objective-C 는 nested namespace 를 가지지 않는다. 그러므로 C++ namespace 내에 Objective-C 클래스를 선언할 수 없고, Objective-C 클래스 내에 namescape 를 선언할 수 없다.

Objective-C 의 클래스, 프로토콜, 그리고 카테고리는 C++ template 내에서 사용될 수 없다. 반대의 경우도 마찬가지다. 즉 Objective-C 인터페이스, 프로토콜, 그리고 카테고리 안에서 C++ template 을 사용할 수 없다.

하지만 Objective-C 클래스들은 어쩌면 C++ template 패러미터들을 처리할 수 있을지도 모른다. C++ template 인자들은 Objective-C 메시지를 표현할 때, receiver 나 인자로써 사용될 수 있다. 단 selector 로는 사용될 수 없다.

9.2 C++의 keyword 때문에 생기는 모호성과 충돌 (C++ Lexical Ambiguities and Conflicts)

여기서는 Objective-C 에 정의된 keyword 을 살펴보고 C++의 keyword 를 사용할때 고려해야할 점에 대해서 알아보겠습니다 .

Objective-C 프로그램이 반드시 알아야 하는 indentifier 가 있는데 그것은 id, Class, SEL, IMP 와 BOOL 입니다.
Objective-C 메소드에서 컴파일러가 self 와 super 를 선언해 놓는데, 이것은 C++가 this 를 그렇게 하는 것과 비슷합니다. 하지만 C++의 this 와는 달리, self 와 super 는 context-sensitive 합니다. 그러므로 Objective-C 메소드 바깥에서는 보통의 identifier 처럼 쓸 수 있습니다 .
이런 context sensitive 한 identifier 로는, 프로토콜과 관련해서는 ***oneway***, ***in***, ***out***, 그리고 ***bycopy*** 가 있습니다. 다른 문맥에서는 이것들은 keyword 로 쓰이지 않습니다.

Objective-C 프로그래머 관점에서 보자면 C++엔 참 많은 keyword 가 생겨났습니다. 이런 C++ keyword 들을 Objective-C 의 selector 로 쓸 수 있습니다. 그러므로 뭐 그렇게 keyword 충돌이 날 가능성은 많지는 않습니다. 하지만 Objective-C 클래스나 인스턴스 변수를 만들기위해서 그런 keyword 를 쓸 수는 없습니다. 예를 들자면 class 는 C++ keyword 인데, 여전히 NSObject 의 메소드인 class:를 쓸 수있습니다.

```
[foo class]; // OK
```

하지만 class 는 keyword 이기 때문에 변수의 이름으로 쓸 수는 없습니다.

```
NSObject *class; // Error
```

Objective-C 에서 클래스의 이름과 카테고리의 이름은 별개의 namespace 에 있습니다. 그러므로 @interface foo 와 @interface (foo)를 같은 소스 코드에 쓸 수 있습니다. Objective-C++에서는 C++의 클래스나 구조체 이름과 같은 카테고리 이름을 사용할 수 있습니다.

프로토콜과 template 를 정의할때는 같은 문법을 사용합니다.

```
id<someProtocolName> foo;
```

```
TemplateType<SomeTypeName> bar;
```

두개가 서로 아주 똑같기 때문에, 컴파일러는 id 를 template 이름으로 사용하는 것을 허락하지 않습니다.

마지막으로 C++에서 전역적인 이름을 갖는 것 앞에 label 이 있을 때, 모호성이 있습니다.

```
label: ::global_name = 3;
```

이 경우에 label 뒤의 콜론 뒤에 빈칸을 반드시 넣어야 합니다. Objective-C++도 비슷한 경우가 있는데, 역시 빈칸을 반드시 넣어야 합니다.

```
receiver selector: ::global_c++_name;
```

Runtime System

이제까지 Objective-C 언어에 대해서 알아보았습니다. 보통의 컴파일러 언어같으면 앞에까지 알게 되면 책을 덮어도 될 것입니다. 그런데 Objective-C 문서엔 꼭 runtime system 에 대한 언급이 나와 있습니다. Java 나 Visual Basic 같은 것도 아니고, 가상 머신에서 돌아가는 것도 아닌 컴파일되는 언어가 runtime system 이라니.. 좀 의아합니다.

앞에서도 계속 언급되는 말이지만 Objective-C 는 동적인 언어입니다. C/C++은 compiler 에서 거의 모든 것을 해결하고 runtime 시에는 그냥 수행되기만 하면 됩니다. 하지만 Objective-C 로 작성된 코드는 C/C++이 컴파일시에 하던 많은 일들을 runtime 시에 합니다. 동적이란 말은, 수행되는 도중에 뭔가 바뀔수 있고, 바뀌어 간다라는 말입니다. 이것을 극명하게 보여주는 것이 id 이며, protocol 등도 runtime 시에 그 대상이 바뀔 수 있다는 것을 앞에서 느끼셨을 것입니다.
그러므로 이런 일을 처리해 줄 수 있는 것이 필요한데, Objective-C runtime system 이 그런 일을 해 줍니다.

Objective-C 로 만들어진 프로그램은 runtime 시스템과 세가지 수준에서 상호 관계를 합니다.

- Objective-C 소스 코드 : runtime 시스템은 뒤에서 자동으로 동작하는데, 코드를 작성하고 컴파일할대 runtime 이 관여합니다.
- Foundation 프레임워크의 NSObject 클래스에 정의된 메소드를 통해서 : 대부분의 Cocoa 객체들은 NSObject 로부터 상속되어 나온 것입니다.

  몇몇 NSObject 메소드들은 runtime system 에 어떤 정보를 요청합니다. 이런 메소드들은 introspection 을 하는데, 이런 예로는 class 메소드가 있습니다. 이것은 한 객체가 자신이 어떤 클래스인지 알 수있게 합니다. isKindOfClass:, isMemberOfClass:, respondsToSelector:, conformsToProtocol:, methodForSelector:같은 것이 그런 예가 되겠습니다.

- runtime 함수를 호출할때 : 이런 함수가 어떤 것이 있는지는 Apple의 Objective-C Runtime Reference를 참조하십시오.

다음의 섹션에서는 NSObject 클래스가 제공하는 프레임웍과 convention 에 대해서 알아보겠습니다. 알아볼 부분을 정리하자면 다음과 같습니다.

- 클래스의 새 instance 를 할당(allocation)하고 초기화(initialization) 하기. 그리고 더 이상 필요하지 않을때 파괴(deallocation)하기
- 다른 객체에 메시지를 전달하기(forwarding)
- 새 모듈을 현재 돌아가고 있는 프로그램에 동적으로 로딩하기

NSObject 의 다른 convention 들은 Foundation framework 리퍼런스에 있는 NSObject 클래스의 스펙에 나와 있습니다.

Chapter 10. 메모리 관리

여기서는 객체를 어떻게 할당하고 초기화 하는지, 그리고 다 쓴 후에 어떻게 폐기시키는지에 대해서 알아보겠습니다.

Cocoa 는 reference counting 이라는 방법을 이용해서 메모리를 관리합니다. 이것은 refcounting 이라고도 짧게 부릅니다. 이 메커니즘에서는 어떤 객체를 쓰려고 하면, 그 객체에 대한 reference counter 를 증가시키고, 다 쓰면 감소시키는 것입니다. 그리고 0 이 되면 해당 객체는 deallocate 됩니다. 이런 방식을 쓰면 안전하게 한 객체의 인스턴스를 다른 객체들이 안전하게 공유할 수 있게 됩니다.

10.1 객체를 할당하고 초기화하기

객체를 만드려면 두 단계를 반드시 거치게 됩니다.

- 동적으로 새 객체를 위한 메모리를 할당하고
- 이렇게 할당된 메모리에 적절한 값으로 초기화를 합니다.

이렇게 해 주지 않으면, 해당 객체는 아직 제대로 작동하지 않게 됩니다. 예를 봅시다.

```
id anObject = [Rectangle alloc];
[anObject init];
```

이것을 한 줄로 줄일수도 있습니다.

```
id anObject = [[Rectangle alloc] init];
```

Objective-C 에서 새 객체를 위한 메모리는 NSObject 에 정의되어 있는 두 클래스 메소드에 의해서 할당됩니다. 다음의 두 메소드가 그것입니다.

```
+ (id) alloc;
+ (id) allocWithZone: (NSZone *) zone;
```

이 두 메소드는 해당 객체의 모든 인스턴스 변수들을 충분히 처리할 만큼 메모리를 할당해 주므로, subclass 에서 override 해서 바꿀 필요가 없습니다.

이 두 메소드는 새로 할당된 객체의 *isa* 인스턴스 변수가 그 객체의 클래스를 가르키도록 초기화 합니다. 다른 instance 변수들은 모두 0 으로 세팅됩니다.

초기화는 class 의 인스턴스 메소드인, init 로 시작하는 메소드들이 합니다. 아무런 인자도 받아들이지 않는 경우엔, init 라는 메소드가 담당합니다. 인자가 있는 경우면 init 라는 prefix 뒤에

argument 에 레이블을 붙여서 사용합니다. 예를들어 NSView 는 initWithFrame: 메소드에 의해 초기화 됩니다.

인스턴스 변수가 있는 모든 클래스는 init..를 제공해야만 합니다. NSObject 는 isa 를 메모리가 할당되었을때 초기화합니다. 그리고 NSObject 의 init 는 self 를 반환합니다.

10.1.1 반환된 객체

init… 메소드는 receiver 의 인스턴스 변수를 초기화하고, receiver 를 반환합니다. 에러없이 사용할 수 있는 객체를 반환하는 것은 init 메소드의 책임입니다.

하지만 때때로 다른 것을 반환해야 할 필요가 있을겁니다. 예를 들어 한 클래스가 여러개의 이름을 가진 객체를 운용하고 있을때, initWithName:을 이용해서 초기화 할텐데, 한 이름당 한개의 객체만이 허용됩니다. 그러므로 같은 이름을 다른 객체에 부여하려고 하면, 그 새 객체를 free 시키고 기존 객체를 반환하게 될것입니다. 즉 이 경우엔 객체가 꼭 self 가 되지는 않는 것입니다.

어떤 경우엔 init… 메소드가 할 일을 제대로 못할 수도 있습니다. 예를 들어 initFromFile: 메소드는 인자로 전해진 파일로부터 필요한 데이터를 얻어오지 못할 수도 있습니다. 전해진 파일이름이 실제로 존재하지 않는다면, 초기화를 하지 못할 것입니다. 이런 경우에 init… 메소드는 receiver 를 free 시키고 nil 을 반환하게 될겁니다. 이럴때의 의미는 "요청한 객체가 만들어지지 않았습니다."하는 의미입니다.

이렇기 때문에 여러분의 코드에서 alloc 에 의해 반환된 객체를 쓰는게 아니라 init 에 의해 반환된 객체를 써야합니다. 예를 들어 다음의 코드는 init 가 반환한 것을 무시하기 때문에 무척 위험한 코드입니다.

```
id anObject = [SomeClass alloc];
[anObject init];
[anObject someOtherMessage];
```

좀더 안전하게 쓰려면, alloc 과 init 를 합쳐서 쓰면 됩니다. 물론 alloc 과 init 사이에 아무것도 쓰지 않는다면 문제가 될 건 없습니다만,이렇게 두개로 나누어 놓으면 그 둘 사이에 다른 코드를 넣을 가능성도 있기에 문제가 될겁니다. 아무튼 요지는 alloc 이 반환한 값을 쓰지 말라는 것입니다. 이렇게하면 좀 더 안전하게 됩니다.

```
id anObject = [[SomeClass alloc] init];
[anObject someOtherMessage];
```

init 가 nil 을 반환할 경우에는, 그것을 검사하는 것이 좋습니다.

```
id anObject = [[SomeClass alloc] init];
if ( anObject )
```

```
        [anObject someOtherMessage];
else
        ...
```

10.1.2 init…의 인자(argument)

init.. 메소드는 객체의 인스턴스 변수들이 뭔가 의미있는 값을 가질 수 있도록 해야합니다. 그렇다고 해서, 꼭 인자를 통해서 모든 필요한 값을 패스할 필요는 없고, 몇몇개는 디폴트 값으로 세팅하거나 0 으로 세팅해도 될겁니다. 그리고 나중에 그 값을 바꿀 수 있는 method 를 제공하면 좋을 것입니다.

NOTE : 주어지는 인자에 따라서 init 메소드의 이름이 바뀐다는 것이 C++ 프로그래머의 관점에선 좀 의아한 것일 겁니다. C++의 constructor 함수는 해당 클래스의 이름과 그 이름이 같으며 단지 인자 리스트 (argument / parameter list)만 바뀔 뿐입니다.
하지만 가만히 보면 결국 Objective-C 도 개념적으론 같습니다. Objective-C 의 경우엔 인자에 label 이 붙어서 결국 그 label 까지 메소드의 이름으로 작용하기 때문에 그렇게 생각될 뿐입니다.

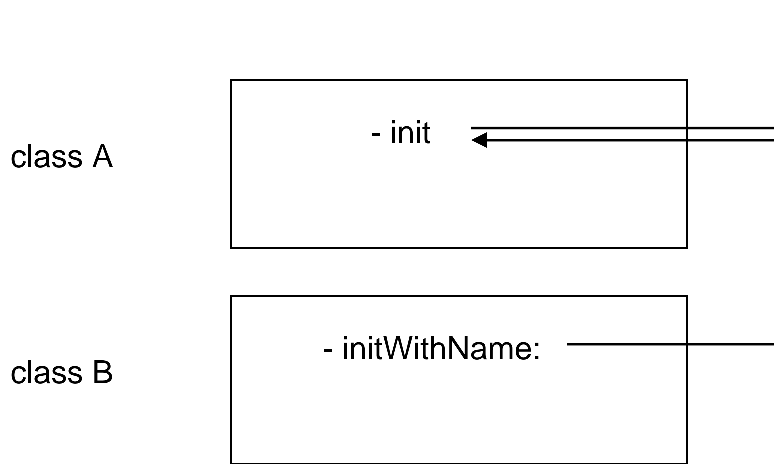
init… 메소드는 적절하지 못한 값이 전달되었을때의 경우도 처리할 수 있도록 만들어져야 합니다.

10.1.3 Class 를 coordination 하기

해당 객체의 init… 메소드는 그 객체의 인스턴스 변수만 초기화 해야 합니다. 상속받은 것에 대해선, super 로 메시지를 보내서 그쪽에서 처리하도록 하는 것이 좋습니다. 예를 봅시다.

```
- initWithName:(char *)string
{
        if ( self = [super init] ) {
                name = (char *)NSZoneMalloc([self zone],
                strlen(string) + 1);
                strcpy(name, string);
                return self;
        }
        return nil;
}
```

super 로 메시지를 보내면 모든 상속된 클래스에 대해서 init 가 연쇄적으로 발생할 수 있게 합니다. 다른 부분보다 제일 먼저 쓰기 때문에, 즉 init… 메소드의 제일 앞에 쓰면, child 클래스들이 초기화 되기 전에 먼저 super class 들을 초기화할 수 있습니다.

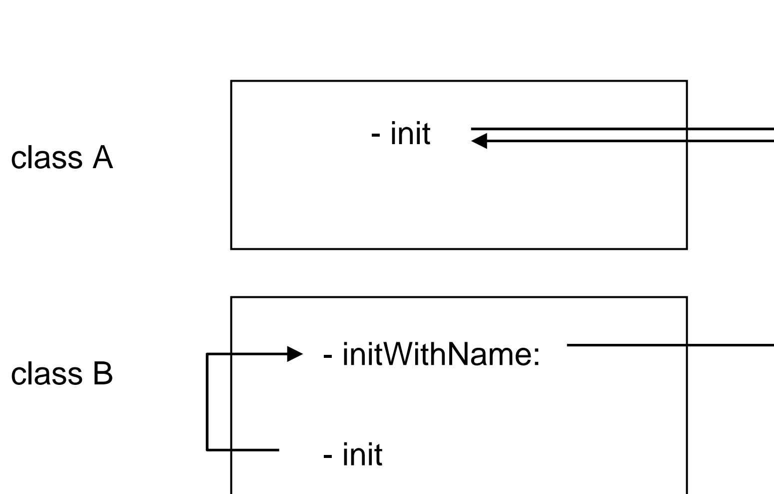


<Fig. 10.1.3 – 1 > chained init

위의 그림은 초기화 과정이 어떻게 서로 연결되어 있는지를 보여줍니다. 그런데 역시 class B 도 다른 클래스가 상속을 할 수 있기 때문에, init 메소드를 넣어 주는게 좋습니다.

```
- init
{
        return [self initWithName:"default"];
}
```

이렇게 하면, 상속한 클래스는 그 super 클래스에 init 메시지를 보내는 것으로 충분히 초기화를 쉽게 할 수 있습니다. 만약 이렇게 하지 않는다면 상속한 클래스는 super 클래스의 init.. method 가 뭐가 있는지 일일이 찾아서 해야 할 것입니다. 이것은 상당히 불편하기도 할 뿐더러, init 를 잘못할 수도 있습니다.



<Fig. 10.1.3 – 2 > covering initWithName with init

10.1.4 Designated Initializer (최종적으로 호출될 initializer)

<Fig. 10.1.3-2>에서 initWithName:은 designated initializer 라고 부릅니다. 즉 그 객체의 초기화 함수로, 최종적으로 불려질 것을 의미합니다. 즉 클래스 B 에서 init 를 호출하면 결국 initWithName:이 호출되므로 "designated"라는 말이 붙여졌겠죠.

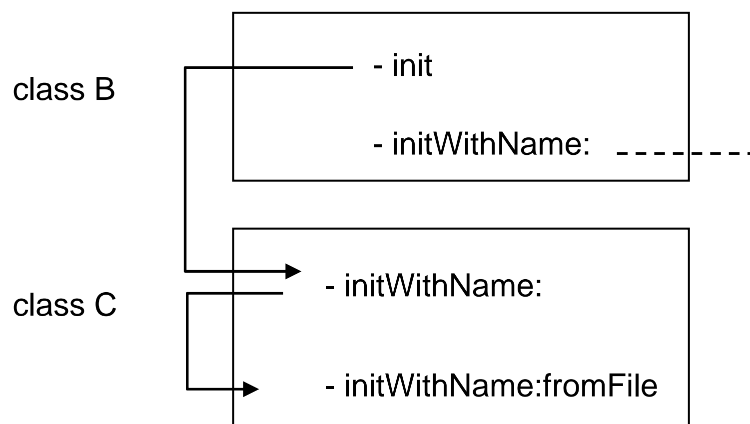
designated initializer 에서는 상속받은 인스턴스 변수들이 제대로 초기화 되도록 super 에게 init 메시지를 보내는 일을 맡게 됩니다. 또한 초기화의 거의 대부분을 하기도 합니다. 그리고 이 밖에 여러 초기화 메소드가 있으면 이 designated initializer 이 그들에 의해서 불려질 최후의 초기화 루틴이기도 합니다. 또 Cocoa 에선 이 designated initializer 에서 해당 객체의 인스턴스가 어떤 종류인지를 결정하는 데이기도 합니다.

꼭 코딩을 이렇게 하지는 않아도 되긴합니다만, 이렇게 하는 것이 convention 이고, Cocoa 의 객체들이 이렇게 만들어져 있으므로, 여러분도 이런 convention 을 따르는 것이, 추후에 기능 확장등을 위해서 좋을 겁니다.

클래스를 정의할때 이런 designated initializer 를 아는게 중요합니다. 예를들어, B 라는 클래스와 그 서브 클래스 C 가 있다고 합시다. C 에는 initWithName:fromFile: 메소드가 있다고 합시다. C 는 상속받은 init 와 initWithName:도 C 를 생성할때 제대로 처리될 수 있도록 해야 합니다. 그렇게 하려면 initWithName:가 initWithName:fromFile:을 호출할 수 있도록 하면 됩니다.

```
- initWithName:(char *)string
{
        return [self initWithName:string fromFile:NULL];
}
```

C 클래스의 인스턴스에선, 상속받은 init 메소드가 이 initWithName 을 호출하게 되고, 그것을 결과적으로 initWithName:fromFile:을 호출하게 됩니다.



<Fig. 10.1.4-1> designated initializer

(Class C 의 인스턴스가 선언이 되어서, Class B 의 초기화 루틴들이 작동될텐데, 그림과 설명이 좀 이상합니다. 더 읽어 봅시다.)

위의 그림은 중요한 부분을 빠트리고 있는데, C 의 designated initializer 인 initWithName:fromFile: 메소드가 super 로 메시지를 보내서 상속받은 초기화 메소드를 호출하게 될겁니다. 근데 B 의 어떤 메소드를 불러야 할까요? init 일까요, 아니면 initWithname:일까요? init 는 아닙니다. 그 이유는 다음과 같습니다.

- 뺑뺑이 호출(circularity)이 발생하게 됩니다. 즉 init 가 C 의 initWithName:을 호출하게 될거고, 그건 initWithName:fromFile:을, 그리고 그것은 다시 B 의 init 를 호출하게 될겁니다.
- B 의 initWithName:을 이용하지 못하게 됩니다.

그래서 initWithName:fromFile:은 B 의 initWithName:을 호출해야만 합니다.

```
- initWithName:(char *)string fromFile:(char *)pathname
{
        if ( self = [super initWithName:string] )
                ...
}
```

NOTE: designated initializer 는 반드시 super 를 통해서 그 super 클래스의 designated initializer 를 불러야만 합니다.

즉 designated initializer 들은 super 를 통해서 서로 연결되어 있게 됩니다. 다른 초기화 메소드들은 self 를 이용해서 각자의 designated initializer 로 연결되게 됩니다.

10.1.5 할당과 초기화를 같이 하기

여기서 설명하려는 것은 앞에서 나온 alloc 과 init 문을 한 줄로 합치는 이야기가 아닙니다. NSString 에 있는 몇가지 메소드 예를 보시면 어떤 것을 의미하는지 알수 있습니다.

```
+ (NSString *)stringWithCString:(const char *)bytes;
+ (NSString *)stringWithFormat:(NSString *)format, ...;
```

위의 함수들은 주어진 인자를 이용해서 NSString 을 만든 후, 그것을 반환합니다. 즉 위의 함수 내부에서는 NSString 객체를 만들고, 전달된 인자를 이용해 초기화한 후 반환합니다. 이런 메소드들에 의해 만들어진 인스턴스들은 **자동으로 deallocation** 됩니다. 그러므로 이런 것들을 쓸 때는 release 를 명시적으로 하지 않아도 됩니다. 단 retain 을 했을 때는, release 를 해야 합니다.

이렇게 할당과 초기화를 같이 하는 함수들은, 할당을 하기 위해서 뭔가 정보가 필요한 경우에 유용합니다. 예를들어, 어떤 객체를 할당하는데, 어떤 내용으로 할지, 몇개나 할지를 파일에서 읽어서

그 안에 있는 정보를 바탕으로 한다고 합시다. 그러면 그런 객체는 listFromFile:과 같은 메소드를 구현해 놓을 것입니다. 그리고 인자로 주어지는 파일을 열어서, 그 안에 있는 내용으로 몇개나 객체를 만들지 알아낸 후, 그런 객체의 List 를 만들어서 반환할 수 있습니다.

또한 이런 방식은 쓰지 않을 새 객체를 위해서 메모리를 할당하는 그런 것을 방지할 수 있는 용도로써도 좋습니다. 앞에서 설명했듯이 init… 메소드는 receiver 와는 다른 객체를 반환할 수도 있습니다. initWithName:이 어떤 이름을 받았다고 합시다. 그런데 그 이름을 위한 객체는 이미 만들어졌다고 합시다. 그러면 이 메소드는 receiver 를 free 시키고 기존의 객체를 반환할 수 있습니다. 이렇게 되면 안쓰는 메모리를 할당할 염려가 없어집니다.
심지어 receiver 를 초기화 해야 하는지를 판단하는 코드를 init.. 메소드에 넣지 않고 allocation 메소드에 넣는다면, 애초에 사용하지 않을 메모리를 할당하는 것 자체를 막을 수도 있습니다. 다음의 예에서 soloist 메소드는 Soloist 클래스의 인스턴스가 하나만 항상 있게끔 해 줍니다.

```
+ soloist
{
	static Soloist *instance = nil;

	if ( instance == nil )
	{
		instance = [[self alloc] init];
	}
	return instance;
}
```

10.2 객체의 소유 관계

Objective-C 프로그램에서는 객체들이 객체들을 생성하고 없앱니다. 한 객체가 자신의 용도에 의해서 다른 객체를 만들고 없애게 됩니다. 그런데, 한 객체가 다른 객체에 메소드를 수행하라고 하면서 뭔가를 전달해 주게 되면, 소유 관계와 그것을 누가 없앨지가 모호해집니다.

보통은 자신이 만들지 않은 것은 없애지 말아야 합니다. Cocoa 에는 그래서 다음과 같은 규칙이 있습니다.

❖ 만약 한 객체가
- 어떤 객체를 만들었다면 (alloc 이나 allocWithZone:을 이용해서)
- 어떤 객체를 카피했다면 (copy, copyWithZone, mutableCopy 혹은 mutableCopyWithZone:을 이용해서)
- 어떤 객체를 계속 가지고 (retain) 있다면 ([retain]을 이용해서)

반드시 그 객체가 [release]를 이용해서 삭제해야 합니다.

❖ 만약 그 객체가 위에서 언급한 객체를 만들거나 카피하지 않았다면, 소유하고 있는 것이 아니므로, release 를 해서는 안됩니다.

한 객체를 만들고 반환하는 메소드를 만들었다면, 그 메소드가 해당 객체를 release 하는 것도 책임지게 해야 합니다. 하지만, 그 객체를 받아서 써야 할 것이 그 객체를 받기 전에, 해당 객체를 despose 해버리면 아무런 쓸모가 없을 것입니다. 그러므로 그럴때는 나중에 release 하라고 마킹해 놓아야 합니다. 그러면 그 객체를 쓰는 것들이 안심하고 쓸 수 있을 겁니다. Cocoa 가 이런 메커니즘을 제공합니다.

10.3 객체를 계속 가지고 있기 (Retaining Object)

동적으로 할당된 객체를 계속 쓰고자 할때는 그것을 지우지 말라고 마킹해 놓는 것이 필요합니다. 이것은 retain 메소드를 이용하면 됩니다. 이것을 이용해서 autorelease 를 pending 시켜놓은 상태로 계속 있을 수도 있습니다. autorelease 에 대해서는 뒤에 "Deallocation" 섹션을 참조하시기 바랍니다. retain 을 하게 되면, 해당 객체를 다 쓰기 전에는 없애지 않게 됩니다. 다음의 예를 봅시다.

```
– (void)setMainSprocket:(Sprocket *)newSprocket
{
        [mainSprocket autorelease];
        mainSprocket = [newSprocket retain]; /* Claim the new Sprocket. */
        return;
}
```

이렇게 하면, mainSprocket 을 다 쓰기 전까지 계속 홀드 해 놓을 수가 있습니다.
만약 다른 객체들과 mainSprocket 을 공유하지 않고, 자체의 용도로 가지고 있기를 원하면 다음과 같이 하면 됩니다.

```
– (void)setMainSprocket:(Sprocket *)newSprocket
{
        [mainSprocket autorelease];
        mainSprocket = [newSprocket copy]; /* Get a private copy. */
        return;
}
```

여기서 주목할 점은, 새 스프로켓과 기존의 mainSprocket 이 같은지 다른지를 검사하지 않는다는 것입니다. autorelease 를 이용하기 때문에 검사할 필요가 없는 것입니다. 만약 autorelease 를 이용하지 않고, 기존의 mainSproket 을 release 하고, 그것과 같은 새것을 쓰려고 할텐데, 이 새것 자체가 없어졌으므로 문제가 생길겁니다. 일일히, 기존의 것을 일단 저장했다가 나중에 release 하는 코드를 만들어도 되지만, 복잡할 겁니다. 예를 들어,

```
– (void)setMainSprocket:(Sprocket *)newSprocket
{
        Sprocket *oldSprocket = mainSprocket;
```

```
        mainSprocket = [newSprocket copy];
        [oldSprocket release];
        return;
    }
```

10.3.1 뺑뺑이 reference (Cyclical References) 다루기

서로를 가르키는 인스턴스 변수가 있는 객체 둘이 있다고 합시다. 둘 사이에 의존 관계는 싸이클 관계, 즉 뺑뺑이 관계를 이루게 됩니다. 예를 들어, 텍스트 프로그램이 있다 합시다. Document 객체는 그 문서 안에 있는 페이지를 가지고 있기 위해 Page 객체를 가지고 있고(retain), 각 page 객체들은 자기가 어떤 문서에 속하는지 알기 위해 Document 객체를 가르키는 인스턴스 변수를 가지고(retain) 있다고 합시다. 사이클 관계에 있기 때문에 상호 의존성이 생깁니다. 그러므로 어느 하나도 release 되지 않게 됩니다. 즉 Document 의 reference counter 는 Page 객체가 다 없어지기전엔 0 이 될수 없고, 그 반대도 마찬가지입니다.

이럴 경우에 해결책은 “parent” 객체는 “children”객체를 retain 하게 하지만, “children”들은 그 “parent’를 retain 할 수 없게 하면 됩니다.

10.4 객체를 없애기(Deallocation)

NSObject 는 dealloc 메소드를 가지고 있는데, 어떤 객체에 할당된 메모리를 release 하고 없앨때 사용합니다. 하지만 이 메소드를 여러분이 바로 부르지는 않습니다. 대신에 release 메소드나 autorelease 를 이용하게 됩니다. 그러면 reference counter 가 줄어들게 되는데, 0 에 도달하게 되면 해당 객체는 없어지게 됩니다.
객체를 deallocation 하는데 있어서 기본 규칙은 다음과 같습니다.

- alloc 이나 copy 를 이용해서 객체의 인스턴스를 얻고 retain 메시지를 보냈다면, 그 인스턴스에 대한 reference 를 가지게 되며, 그것을 이용해서 나중에 다쓰고 난 후 release 를 해야 합니다.
- 멀티쓰레드나 distributed object 상황이 아닌 경우를 제외하고, 앞의 경우와 다른 방식으로 객체에 대한 인스턴스를 얻었다면, 그것을 release 하면 안되며, 현재 method 의 scope 넘어로 가져가서 쓰면 안됩니다. scope 바깥에서도 쓰고 싶으면 retain 메소드를 이용하십시오.
- autorelease 는 “나중에 release 하시오”라는 의미입니다.

10.4.1 공유된 객체를 release 하기

Cocoa 의 소유 정책은 언제 객체를 없애야 할것인지에 대한 정도입니다. 메소드 안에서 받은 객체가 그 메소드의 scope 내에서 항상 유효한지에 대해서는 정해 놓지 않았습니다. 얻은 객체는, 그 소유권을 가진 객체가 release 되면 언제나 무효한 것이 됩니다. 혹은 소유권을 갖은 객체가 없어지지

않는 경우라도, 해당 객체를 가지고 있는 인스턴스 변수를 다른 값으로 재할당해도 무효하게 됩니다. release 외의 메소드들 중 객체를 없애는 것은 반드시 그렇게 한다고 도큐먼트에 써져 있어야 합니다.

예를들어 한 객체의 주된 스프로켓을 받아내고, 그 객체 자체를 업애면, 그 스프로켓도 없어진다고 가정해야 합니다. 비슷하게, 주된 스프로켓을 요청하고 setMainSprocket:을 보내면, 받은 스프로켓이 유효한 상태로 있다고 가정해서는 안됩니다.

```
Sprocket *oldMainSprocket;
Sprocket *newMainSprocket;

oldMainSprocket = [myObject mainSprocket];

/* If this releases the original Sprocket... */
[myObject setMainSprocket:newMainSprocket];

/* ...then this causes the application to crash. */
[oldMainSprocket anyMessage];
```

이런 것을 막으려면 무효화 시킬 만한 메시지를 보내기 전에 retain 과 autorelease 메시지를 이용해야 합니다.

```
Sprocket *oldMainSprocket;
Sprocket *newMainSprocket;

oldMainSprocket = [[[myObject mainSprocket] retain] autorelease];
[myObject setMainSprocket:newMainSprocket];
[oldMainSprocket anyMessage];
```

이렇게 retain 과 autorelease 를 이용하면 oldMainSprocket 이 scope 내에서 계속 유효하게 살아남아 있게 됩니다. 비록 setMainSproket 이 그것을 release 하더라도 말입니다.

10.4.2 인스턴스 변수들을 release 하기

NSObject 의 dealloc 메소드는 receiver 의 인스턴스 변수들을 deallocate 하기는 하지만 다른 메모리를 가르키고 있는 변수를 따라가서 없애진 않습니다. 이것은 흡사 C 에서 free 를 할 때와 마찬가지입니다. 만약 receiver 가 이렇게 추가적인 메모리를 할당받고 있다면, 즉 예를 들자면 문자열이나, 구조체의 어레이 같은 것을 가지고 있다면, 이런 것들은 반드시 deallocation 이 되어야 합니다. 단 다른 뭔가가 동시에 그것을 쓰고 있지 않다면 말입니다.

그러므로 NSObject 의 서브클래스들이 dealloc 을 override 하는게 필요합니다. 즉 이런 추가적인 메모리를 할당받고 있는 모든 클래스는 자체적으로 dealloc 메소드를 가지고 있어야 합니다. 이때 그런 dealloc 메소드는 그 메소드의 제일 마지막에서 super 의 dealloc 을 호출해야 합니다.

참고로 말하자면 객체를 생성할때는, super 클래스의 것부터, 없앨때는 자신의 것부터 없애는 것입니다.

예를 보겠습니다.

```
- dealloc {
        [companion release];
        free(privateMemory);
        vm_deallocate(task_self(), sharedMemory, memorySize);
        [super dealloc];
}
```

10.4.3 나중에 release 하도록 객체에 표시해 놓기

NSObject 에서 정의하고 있는, autorelease 는 그 메시지의 receiver 가 나중에 release 되게 표시를 해 놓는 역할을 합니다. 어떤 객체에 autorelease 메시지를 보내면, 그 말은 곧 "autorelease 가 보내진 곳의 scope 를 넘어가서는 그 객체가 필요하지 않아"라는 뜻입니다. 그럼 실제로 삭제가 된느 시점은 언제일가요? 여러분의 코드가 완전히 끝나고, 제어권이 프로그램의 객체에 돌아오면, 즉 event loop 이 끝나는 시점에, 프로그램이 그 객체를 release 합니다. 앞에서 예를 들어본 sprockets 메소드는 다음과 같이 구현될 수 있겠습니다.

```
– (NSArray *)sprockets
{
        NSArray *array;

        array = [[NSArray alloc] initWithObjects:mainSprocket,
                                        auxiliarySprocket, nil];

        return [array autorelease];
}
```

다른 메소드가 위의 sprocket 어레이를 받았을때, 그 어레이가 필요치 않게되는 시점에 없어지리라고 가정할 수 있습니다. 하지만 그 scope 안에선 안심하고 계속 쓸 수 있습니다. 단 공유된 객체와 관련될 때는 그렇지 않을 수도 있습니다. 심지어 그 어레이를 저 method 를 부른 객체에 반환할 수도 있습니다. 그러므로 autorelease 메소드는 모든 객체들이 동적으로 할당된 객체를 없애야 할지 어떨지 걱정하지 않고 쓸 수 있도록 해 줍니다.

NOTE : 이미 없어진 객체에 자꾸 release 메시지를 보내면 에러가 나듯이, autorelease 를 자꾸 보내도 에러가 납니다. release 나 autorelease 를 보내는 횟수는 다음으로 제한됩니다.

release/autorelease 보내는 횟수 = create (1) + the number of retain message sent

Chapter 11. 메시지 forwarding

기본적으로 처리할 수 없는 메시지를 보내는 것은 에러입니다. 하지만 runtime 시스템은 그 메시지를 받는 객체에 한번의 기회를 줍니다. 즉 forwardInvocation:이라는 메시지를 NSInvocation 객체를 인자로 해서, runtime 시스템이 그 받는 객체에 보내게 되는데, NSInvocation 객체는 원래의 메시지와 인자들을 가지고 있게됩니다.

해당 메시지에 대해서 디폴트 응답을 부여하기 위해서 forwardInvocation: 메소드를 구현할 수 있습니다. 혹은 좀 다른 방식으로 에러를 막기 위해서 만들수도 있을 겁니다. 이름에서 보면 알 수 있듯이 그 메시지를 다른 객체에 보내기 위해서 보통 사용합니다.

언제 이런 메시지 포워딩을 사용할지 생각해 봅시다. negotiate 란 메시지에 반응하는 객체를 만드다합시다. 그리고 그 반응에 다른 객체로부터의 응답도 포함시키길 원한다고 합시다. 그러면 여러분이 구현하고 있는 negotiate 메소드에 이 negotiate 메시지를 그 다른 객체에 전달하는 부분을 넣으면 됩니다.

만약 여러분의 클래스가 negotiate 메소드를 상속받지 못하는 상황이래도, 빌려 쓸 수가 있습니다. 즉 해당 메시지를 다른 클래스의 인스턴스에 전달하도록 만들면 됩니다.

```
- negotiate
{
        if ( [someOtherObject respondsTo:@selector(negotiate)] )
                return [someOtherObject negotiate];
        return self;
}
```

아무튼 forwardInvocation: 메시지에 의해 얻은 두번째 기회는, 정적이라기 보다는 동적인 해결책을 제시해 줍니다. 즉 이렇게 움직입니다. 한 객체가 주어진 메시지에 반응을 못하면, runtime system 은 그 객체에 forwardInvocation 메시지를 보냅니다. NSObject 클래스의 서브 클래스들은 이 메시지를 상속받습니다. 하지만 NSObject 의 이 메소드는 doesNotRecognizeSelector:를 호출할 뿐입니다. 그러므로 이 것을 overriding 해서 여러분 자신의 것을 구현하면 forwardInvocation:메시지를 받았을때, 다른 객체에 메시지를 전달하도록 할 수 있습니다.
포워딩을 할때, forwardInvocation: 메소드가 해야 할 일은 다음과 같습니다.

- 메시지가 보내질 곳을 파악하고
- 원래의 인자를 유지한채 그대로 보내기

```
- (void)forwardInvocation:(NSInvocation *)anInvocation
{
        if ([someOtherObject respondsToSelector: [anInvocation selector]])
```

```
        [anInvocation invokeWithTarget:someOtherObject];
    else
        [super forwardInvocation:anInvocation];
}
```

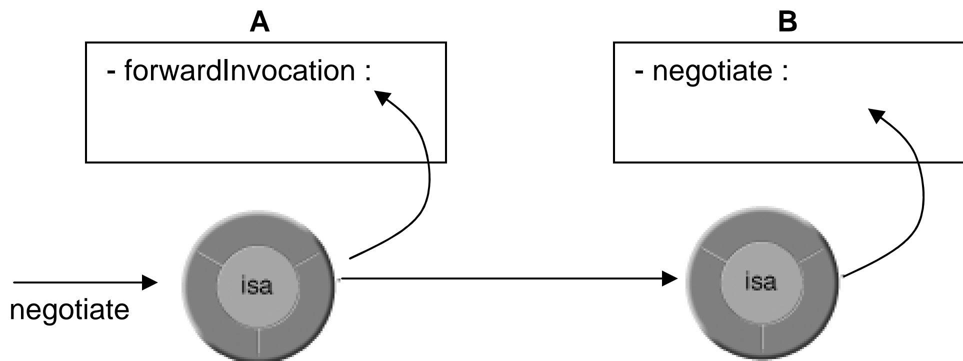
포워딩된 메시지에 대한 응답은 애초의 발송자한테 반환됩니다. 그리고 그 타입도 모든 종류가 다 됩니다.

이 메커니즘을 이용하면 어떻게 처리해야 할지 모르는 메시지들을 분배해주는 센터로써 움직일수있게 합니다. 혹은 잠시 머물러있다 가는 곳으로 만들 수도 있습니다. 다른 메시지로 번역을 해 줄수도 있고, 때로는 그냥 메시지를 먹어 버리는 수도 있습니다. 또한 여러 메시지를 다른 하나의 메시지로 낼 수도 있습니다. 이 메소드가 뭐를 할지는 구현하기 나름입니다.

NOTE : 포워딩을 하는 객체안에 이미 그 포워딩하려는 메시지를 처리하는 메소드가 있다면 절대 다른 객체로 포워딩이 되지 않습니다. 예를 들어 이미 어떤 객체가 negotiate 라는 메소드를 가지고 있고, 그 메시지를 받아서 다른 객체에 포워딩 시키려하면 전달되지 않고, 자체에서 처리가 됩니다.

11.1 전달(Forwarding)과 다중 상속(Multiple Inheritance)

포워딩은 마치 상속(inheritance)과 흡사하게 되어 있습니다. 그래서 때로는 Objective-C 에서 다중 상속의 효과를 내기 위해서 쓸 수도 있습니다. 다음 그림에서 보듯이 한 메시지를 다른 데로 전달해서 그 메시지에 대한 반응을 보이는 객체는 흡사 다른 클래스에 정의된 메소드를 빌려오거나 상속받은 것처럼 보입니다.



<Fig. 11.1-1> 메시지 포워딩

클래스 A 가 netotiate 라는 메시지를 받으면 클래스 B 에 보내고, 클래스 B 는 그 결과를 다시 A 에게 보냅니다. 그리고 그것을 다시 A 는 원래 메시지를 보낸 측에 반환합니다. 그러므로 흡사 외부에서 볼때는 클래스 A 가 negotiate 라는 메시지를 처리하는 것처럼 보입니다.

그러므로 A 는 자체 상속관계에서 오는 상속과 더불어 B 의 메소드를 상속한 것처럼 행동합니다. 그러므로 다중 상속처럼 쓸 수가 있습니다.

하지만 이런 상속과 메시지 포워딩은 근본적인 차이점이 있습니다.

다중 상속	메시지 포워딩
여러 클래스로부터의 메소드나 멤버 변수를 한 클래스에 몽땅 집어 넣어, 결과적으로 비대해진 클래스가 됩니다.	자신은 꼭 자신이 처리할 것만 처리하고, 다른 객체가 처리할 수 있는 것은 각각의 객체들이 처리할 수 있도록 합니다. 그러므로 그 자신은 다중 상속의 경우처럼 큰 객체가 되지 않습니다.

<표 11.1-1> 다중 상속과 메시지 포워딩의 차이

11.2 대리 객체(Surrogate Object)

이 대리 객체는 어떤 일을 대신해주는 객체입니다. 이 개념은 바로 앞에서 설명한 메시지 포워딩에서 어떤 메시지를 자신이 처리하지 않고, 그것을 처리할 수 있는 객체에 넘겨준다는 것을 생각해 보면 이해하기 쉽습니다. 즉 그 전달해 주는 객체를 대리 객체라고 할 수 있는 것입니다.

어떤 것이 어떤 것을 대신한다고 하는 것은 사실 생각하기 나름이지만, 직접적으로 해당 메시지를 대신 받아서 넘겨주기때문에 "대리"한다고 하는 것으로 보입니다.

다음 섹션인 "원격 메시징"에서 나올 proxy 가 바로 이런 대리 기능의 예가 되겠습니다. 즉 proxy 객체는 실제로 주어진 메시지를 처리하는 객체에 그 메시지를 넘기기 전에, 넘어온 패러미터가 제대로 잘 되어 있나 등을 살펴 봐 줄 수 있습니다.

다른 예도 있을 수 있겠습니다. 예를 들어 많은 데이터를 처리해야 할 객체가 있는데, 그런 데이터는 사이즈가 큰 데이터 파일, 이미지 등으로 구성되어 있다고 합시다. 혼자서 이런 일을 하기엔 상당히 시간이 걸릴겁니다. 그러므로 당장 지금 필요한게 아니라면 그런 부분은 나중에 처리할 수 있음 좋을 겁니다. 당장은 모든 데이터들이 위치할 공간만 제공하면 됩니다. 자 이런 경우에, 맨처음엔 모든 기능을 가진 객체를 만드는게 아니라 작은 크기의 꼭 필요한 기능만 가진 객체를 만들고서, 일단 당장 필요한 일들을 하다가, 나중에 정말 큰 데이터를 처리해야할 필요가 있을때, 그때 적절한 객체를 만들어서 그 객체가 처리할 수 있도록 하는 것입니다.

11.3 전달과 상속

전달(forwarding)이 상속을 흉내내기는 하지만, NSObject 는 두 개념을 구별합니다. 이를테면 respondToSelector:나 isKindOfClass:는 상속 관계만 보고 답을 주지 forward chain 을 보고 답을 주지는 않습니다. 예를 들어 앞의 클래스 A 가 negotiate 라는 메시지에 반응하는지 알아보려고 다음과 같이 했다고 합시다.

```
if ( [aWarrior respondsToSelector:@selector(negotiate)] )
        ...
```

그러면 오는 결과는 NO 입니다. 클래스 A 가 물론 그 메시지를 제대로 다른 것에 전달해서 처리를 하는데도 말입니다.

그러므로, 대개의 경우에 NO 라고 답이 오면 해당 클래스가 그 메시지를 처리 못하겠지만, 꼭 그런것은 아닙니다. 만약 이런 클래스가 respondToSelector 에 YES 라는 반응을 하게끔 만들고 싶으면 다음과 같이 overriding 을 해야 합니다.

```
- (BOOL)respondsToSelector:(SEL)aSelector
{
        if ( [super respondsToSelector:aSelector] )
                return YES;
        else {
                /* Here, test whether the aSelector message can *
                * be forwarded to another object and whether that *
                * object can respond to it. Return YES if it can. */
        }
        return NO;
}
```

isKindofClass 의 경우도 마찬가지입니다. 그리고 이외에도 instancesRespondToSelector 와 같은 메소드도 포워딩 메커니즘을 고려해서 override 해야 합니다. 만약 프로토콜이 사용된다면 conformsToProtocol:도 같은 식으로 해주어야겠습니다. 그리고 메시지가 온 것을 다른 객체에 전달하는 객체는 반드시 methodSignatureForSelector:라는 메소드를 가지고 있어야 합니다. 이 메소드는 전달한 메시지에 최종적으로 반응을 보여줄 메소드에 대한 설명을 반환해야 합니다.

어쩌면 전달 알고리즘을 private code 에 넣고 forwardInvocation:을 포함한 이 모든 메소드들이 그 전달 알고리즘을 호출하도록 할수도 있습니다.

NOTE : 이 방법은 다른 어떤 방법으로도 해결이 안되는 특수한 상황에서 쓰는 고급 기법입니다. 이것은 상속을 대치하려는 목적으로 만들어진 것은 아닙니다. 만약 이 기법을 꼭 써야만한다면, 전달하는 클래스가 어떻게 움직이는지와 메시지를 전달받는 클래스가 어떻게 움직이는지를 완전히 이해할때만 하십시오.

이 기법에 대한 더 자세한 설명은 Foundation framework 리퍼런스의 NSObject 클래스 스펙에 나와 있습니다. invokeWithTarget 에 대한 설명은 역시 같은 리퍼런스의 NSInvocation 클래스 스펙을 보십시오.

Chapter 12. 동적 로딩(Dynamic Loading)

Objective-C 프로그램은 돌고 있는 동안에 새 클래스와 카테고리를 로딩하고 링크할 수 있습니다. 이것은 DLL 이나 Shared library 와는 좀 다른 개념입니다. 차라리 플러그인과 비슷하다고 생각하면 되겠습니다. 이렇게 로딩된 코드는 돌고 있는 프로그램에 속하게 되면서 애초에 로딩된 클래스나 카테고리와 완전히 같은 식으로 취급됩니다.
이런 것이 사용되는 예는 System Preference 을 열면 보이는 프로그램들을 들 수가 있습니다.

Cocoa 환경에서 이런 동적 로딩을 사용하는 예는 프로그램을 커스터마이즈하도록 할 때 보통 쓰입니다. 즉 다른 사람들이 여러분의 프로그램이 런타임시에 로딩할 수 있는 모듈을 만들 수 있습니다. 이런 예는 Interface Builder 의 경우가 되겠습니다. 즉 커스텀 팔렛을 Interface Builder 는 이런 식으로 띄웁니다. Mac OS X 의 System Preference 에도 이런 경우가 있습니다. 이렇게 모듈을 어느때나 로딩이 가능하게 하면 이미 만들어진 여러분의 프로그램에 기능을 추가할 수 있습니다. 즉 여러분이 프로그램을 만들때 허가한 선내에서, 하지만 결코 기대하거나 이렇게 저렇게 하라고 정의하지 않은 방식으로 확장을 할 수가 있습니다. 여러분은 framework 를 제공하고 다른 사람들은 코드를 제공하는 것입니다.

비록 Mach-O 파일에 있는 Objective-C 모듈을 동적으로 로딩하는 런타임 함수가 있기는 하지만, Cocoa 의 NSBundle 클래스 자체에서 아주 편리한 메커니즘을 제공합니다. Foundation framework 리퍼런스의 NSBundle 에 대한 스펙을 보시면 더 자세한 내용이 나와 있습니다. Mach-O 에 대한 내용은 Mach-O Runtime Architecture 를 살펴보시기 바랍니다.

Chapter 13. 원격 메시징(Remote Messaging)

원래 Objective-C 가 한 어드레스 공간에서 단 한개의 프로세스로 실행되는 프로그램을 만들기 위해서 만들어지긴 했지만, OOP 모델 자체가 프로세스간 통신(interprocess communication)에도 잘 맞습니다. 다른 주소 공간에 있는, 즉 다른 태스크를 위한 객체간에, 혹은 같은 태스크라도 다른 thread 를 위한 객체간에 Objective-C 메시지가 왔다 갔다 하는 모습은 쉽게 상상할 수 있습니다.

NOTE : Objective-C 가 사용하는 OOP 개념은 SIMULA 에서 나온 C++이 사용하는 개념과 비슷하지만 이런 점에서 좀 다릅니다. 물론 C++에서도 distributed object 라는게 있지만, 클래스의 syntax 자체를 놓고 봤을때, 그냥 method call 이지 message passing 같은 느낌이 잘 안들게 되어 있기 때문입니다. 실제로 두 언어를 위한 입문서들을 열어보면 그런 느낌을 확연히 받을 수 있습니다. Objective-C 는 message passing 이라고 주로 하고 C++에서는 member function call 이라는 표현이 주가 되는 것을 보면 알 수 있습니다. 그러므로 같은 OOP 라도 접근이 좀 달라질 수 있고, 그것은 프로그램의 구조에 영향을 미칠 수 있다는 점을 염두에 둡시다.

예를 들어 전형적인 클라이언트-서버 모델에서, 클라이언트는 서버측에 객체를 보내고, 서버는 notification 이나 그밖의 다른 정보에 대한 것을 객체화 해서 클라이언트에 보낼 수 있습니다.
혹은 같은 도큐먼트를 수정하는 여러 프로세스를 생각할 수도 있겠습니다. 각 태스크는 한 도큐먼트에 있는 각각의 데이터 타입을 위해서 만들어져 있을 수있겠습니다. 예를 들어, 그림은 이 태스크가, 글은 저 태스크가, 소리는 또 다른 태스크가 담당하는 식으로 말입니다. 이때 한 태스크는 그것을 통합해서 화면에 보여주고, 다른 여러 태스크에 사용자가 명령한 것을 분배 해 줄 수있겠죠. 그 태스크 사이에 Objective-C 메시지가 왔다 갔다 할 수 있습니다.

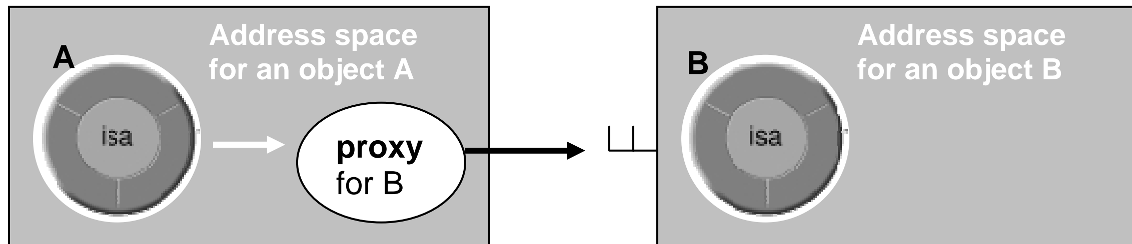
13.1 분산 객체 (Distributed Objects)

Objective-C 에서 원격 메시징을 하려면 runtime 시스템이 다른 주소 공간에 있는 객체들을 서로 연결 시켜주어야 하고, 멀리 떨어진 주소 공간에 있는 객체에 언제 메시지가 전달될지를 파악해 주어야 합니다. 또한 두 분리된 태스크간에 스케쥴링을 처리해 주어야 합니다. 예를 들자면 받으려하는 객체가 하던 일을 다 끝내고 새로운 것을 처리할 준비가 되었을때까지 다른 객체가 보낸 메시지를 당분간 가지고 있던가 하는 그런 것을 말합니다.

Cocoa 는 분산 객체 구조를 포함하고 있는데, 이것은 기본적으로는 runtime 시스템에 대한 일종의 extension 입니다. 분산 객체를 이용하면 다른 태스크에 있는 객체에 메시지를 보내거나, 같은 태스크를 수행하는 다른 thread 에 보낼 수도 있습니다. 이런 thread 의 경우는 비록 같은 태스크를 수행하고 있는 상이한 thread 라 할지라도 다른 태스크를 수행하는 thread 인양 취급합니다.
Cocoa 의 분산객체 시스템은 runtime 시스템 위에 만들어졌다는 것을 명심하십시오. 그러므로 Cocoa 객체의 기본적인 습성을 바꾸진 않습니다.

원격으로 메시지를 보내려면, 그 프로그램은 일단 그 떨어져 있는 receiver(remote receiver)와 통신 체계를 우선 구축해야 합니다. 그럴려면 우선 그 원격 객체에 대한 proxy 를 메시지를 보내는

측과 같은 주소 공간에 만들어야 합니다. 일단 이렇게 해놓으면 원격 객체와의 통신은 그 proxy 를 통해서 이루어집니다. proxy 는 말 그대로 해당 원격 객체인양 행동을 하게 됩니다. 하지만 그 자체로서 identity 는 가지지 않습니다. 프로그램은 이 proxy 를 마치 해당 원격 객체인양 취급합니다.여러분은 이 proxy 를 그냥 그 원격 객체이거니 하고서 다루면 됩니다.
이것을 도식화 하면 다음과 같습니다.



<Fig. 13.1-1> Proxy 와 분산 객체

sender 와 receiver 는 다른 태스크에 있고, 스케쥴링도 따로 됩니다. 그래서 sender 가 메시지를 보낼때, receiver 가 항상 받을 준비가 되어 있다고 생각하기는 쉽지 않습니다. 그래서 받아지는 메시지는 일단 큐에 들어가게 됩니다. 그리고 receiver 가 받을 준비가 되면 그 큐에서 하나씩 빼서 보게 됩니다.

Proxy 가 하는 것은 원격 객체를 대신하는 것은 아니고, 그 원격 객체를 억세스하는 것도 아닙니다. 즉 원격 객체의 한 카피가 아니라, 단지 메시지를 그 원격 객체에 보내고 그 통신을 관리하는 것입니다. 즉 주된 일은 원격 객체에 대한 로컬 주소를 제공하는 것입니다. 그렇다고 완전히 투명성이 있는 존재는 아닌데, 즉 원격 객체의 인스턴스 변수들을 바로 가져오거나 세팅할 수 없습니다.

보통 원격지에 있는 메시지를 받는 객체는 무명(無名) 혹은 익명(溺名)이겠습니다. 즉 그 클래스 자체는 원격 프로그램 내부에 숨겨져 있게 됩니다. 메시지를 보내는 측에서는 받는 측에 대해서 자세한 정보는 알 필요가 없고, 단지 해당되는 원격 객체가 어떤 메시지에 반응하는 가만 알면 됩니다.

이런 이유로, 원격 메시지를 받는 객체는 자신의 인터페이스를 formal protocol 로 공개해야 합니다. 보내는 측과 받는 측이 다 그 protocol 을 선언해야합니다. 즉 둘다 같은 protocol 선언을 받아들이면 됩니다. 받는 측이 선언을 해야 하는 이유는, 원격 객체가 반드시 그 프로토콜을 준수해야 하기 때문입니다. 보내는 측은 컴파일러에게 그 메시지에 대한 정보를 주고, conformsToProtocol:을 사용할 필요성이 있고, 또한 원격 수신자를 테스트하기 위해 @protocol() 디렉티브를 사용해야 하기 때문에 선언해야 합니다. 하지만 송신자는 그 프로토콜에 대한 구현은 가지고 있지 않아도 됩니다. 단지 선언만 하면 충분합니다. 왜냐하면 실지로 그것을 송신자측에서 실행하는게 아니기 때문입니다.

NSProxy 와 NSConnection 을 비롯한 분산 객체 구조에 대해선 Foundation framework 리퍼런스에 잘 나와 있으니 참조하시기 바랍니다.

13.2 언어에서 받는 지원

원격 객체는 프로그램 디자인시에 흥미로운 디자인을 할 수 있게 해주지만, 또한 Objective-C 언어 자체에 몇가지 흥미로운 사항을 제기 시켜줍니다. 이런 문제 중 대부분은 리모트 메시징의 효율성과 이런 원격 메시징에 가담하는 두 태스크를 얼마나 분리 시켜줄까 하는 문제들과 관련되어 있습니다.

리모트 메시징에 쓰기 위한 용도로써 몇가지 명령어를 사용할 수가 있는데, Objective-C 는 6 개의 다음과 같은 타입을 정해 놓았습니다. 이 타입들은 formal protocol 내에서 메소드를 선언하는데 쓰입니다.

- oneway
- in
- out
- inout
- bycopy
- byref

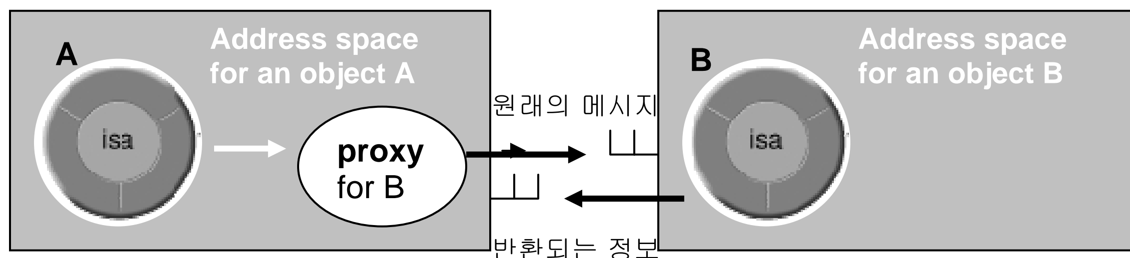
이 모디파이어들은 formal protocol 에서만 사용하십시오. 즉 클래스나 카테고리 선언에서는 사용할 수가 없습니다. 하지만 만약 클래스나 카테고리가 프로토콜을 받아들이면, 프로토콜 메소드를 구현하는 부분에서는 선언할때 쓰였던과 같은 모디파이어를 쓸 수가 잇습니다.

13.2.1 동기 메시지와 비동기 메시지(Synchronous and Asynchronous Messages)

우선 단순한 리턴 값을 가지는 메소드를 생각해 봅시다.

```
- (BOOL) canDance;
```

같은 프로그램에 있는 수신측에 canDance 메시지를 보내면, canDance 메소드가 수행되고서 송신자에게 바로 리턴값을 반환합니다. 하지만 수신측이 원격 프로그램에 있으면, 두개의 기본적인 메시지가 필요합니다. 즉 하나는 원격 메소드를 호출하기 위해서 원격 객체를 얻어내는 메시지이고, 다른 하나는 원격으로 계산한 결과를 다시 반환하는 메시지입니다. 다음의 그림을 봅시다.



<Fig. 13.2.1-1> Rount-Trip Message

보통 리모트 메시지들은 위의 그림에서 볼 수있는 것과 같은 양방향 혹은 왕복 remote procedure call (RPC)입니다. 송신자 측은 메시지를 수신자측에 보내기 위해, 수신자측의 프로그램을

기다리다가 보내고, 수신자가 프로세싱하던 것을 다 마치면 그렇다고 송신자측에 indication 을 애초에 요청받았던 리턴 값과 함께 보냅니다. 정보가 반환되지 않더라도 수신자가 끝낼때까지 기다리는 것은 두 프로그램이 동기를 하는데 도움이 됩니다. 이런 이유로 보통 왕복 메시지는 synchronous 하다고 말합니다. 원격 메시징의 기본 세팅은 이런 동기 메시징입니다.

하지만 응답이 올때까지 기다리는게 항상 좋다고만은 할 수 없습니다. 때때로 원격 메시지를 dispatch 하고 자기 할 일을 하는게 더 바람직할 수도 있습니다. 그러면 수신자는 메시지를 받아서 처리하는 동안 송신자는 그외에 자기가 할일을 할 수가 있습니다. 이런 경우가 바로 비동기 메시징 (asynchronous messaging) 인데 oneway 라는 모디파이어를 쓰게 됩니다.

```
- (oneway void) waitzAtWill;
```

oneway 가 비록 const 처럼 타입을 정의하는 것이긴 하지만, 다른 타입 이름과 같이 쓸 수도 있습니다. 즉 oneway float 나 oneway id 같이 말입니다. 임의의 타입을 지정할때는 oneway void 라고 하면 되겠습니다. 이런 비동기 메시지는 리턴값을 가지지 않습니다.

13.2.2 포인터 인자(Pointer Arguments)

이젠 포인터 인자를 갖는 메소드를 생각해 봅시다. 포인터를 이용하면 pass-by-reference 를 이용해서 정보를 넘길 수 있습니다.

```
- setTune:(struct tune *)aSong
{
        tune = *aSong;
        ...
}
```

정보를 리턴할 때도 마찬가지겠죠.

```
- getTune:(struct tune *)theSong
{
        *theSong = tune;
        ...
}
```

포인터를 쓰면 리모트 메시지가 실행되는 방식에 차이를 만들수가 있습니다. 어떤 방식이 되던간에 수신측에 전달되면 포인터가 바뀌게는 됩니다. 즉 포인터는 송신자측의 주소 공간에 있는 위치를 가르치게 되어서 수신측에서는 의미가 없게 됩니다. 그러므로 runtime 시스템이 여기에 어떤 조정을 가하게 됩니다.

pass-by-reference 를 이용하는 인자를 쓰면, runtime 시스템은 그 포인터를 dereference 하게 됩니다. 즉 그 내용을 까보게 된다는 것입니다. 그리고 그 포인터가 가르키고 있는 위치의 내용을

원격 프로그램에 보냅니다. 그리고 그 원격 프로그램의 로컬 주소 공간에 저장하고, 그 주소를 원격 프로그램에 전달합니다. 그러므로 송신측과 수신측의 pointer 가 가르키고 있는 메모리 위치는 달라집니다.

반면에 pass-by-reference 스타일로 정보를 리턴할때는, 수신측의 포인터가 가르치는 메모리 위치가 송신측에 보내지진 않습니다. 단지 그 내용이 송신측에서 애초에 수신측에 보낼때 사용했던 그 메모리 위치에 들어가게 됩니다.

이런 메커니즘을 위해서 런타임 시스템이 송신/수신에 대해서 다른 행동을 취해야 하므로, Objective-C 는 프로그래머가 의도를 명확하게 표시하도록 타입 모디파이어를 제공합니다.

- in : 정보가 메시지에 포함되어서 전달되게 합니다.
  - setTune:(in struct tune *)aSong:
- out : 해당 인자가 pass-by-reference 로 정보를 반환할때 쓰이도록 합니다.
  - getTune:(out struct tune *)theSong:
- inout : 해당 인자가 정보를 제공하거나 반환할때 쓰이도록 합니다.
  - adjustTune:(inout struct tune *)theSong:

NOTE : 이 설명을 보면 in/out/inout 은 수신자의 관점에서 만들어졌다는 것을 알 수있습니다. 수신자 측에서 보자면 in 이 정보를 받는거고 out 이 정보를 리턴하는 것이니까 말입니다. 구현이 수신측에 들어가지 송신측에 들어가지 않는다는 것과 맥락을 같이 한다고 볼 수 있습니다.

Cocoa 분산 객체 시스템은 대부분의 포인터 인자는 디폴트로 *inout* 을 사용합니다. 하지만 *const* 로 선언된 것은 *in* 을 디폴트로 사용합니다. inout 이 물론 별 걱정안하고 안전하게 사용할수 있지만 처리 시간이 많이 걸립니다. 왜냐하면 양방향으로 정보를 보내야 하기 때문입니다. pass-by-value 즉 포인터가 아닌 변수를 인자로 할때 쓸 수 있는 것은 *in* 뿐입니다. 즉 *in* 은 어떤 형태의 인자에도 사용할 수 있지만 *out* 과 *inout* 은 포인터인 경우만 사용됩니다.

C 언어에서는 포인터를 이용해서 여러 값을 하나로 뭉뚱그려서 사용하곤합니다. 예를 들면 스트링은 캐릭터 포인터로 되어서, 여러 캐릭터를 한 뭉치로 만들어서 쓰는 것입니다. 물론 표현법과 실제 구현에서 어느 정도 우회적인데가 있지만, 개념상으로는 그렇지 않습니다. 결과적으로 스트링은 한개의 단일개체이지, 다른 뭔가를 가르키기 위한 포인터는 아닙니다.

마찬가지로 분산 객체 시스템에서는 포인터를 자동으로 dereferencing 해서 그게 가르키는 것을 value 인양 전달해 줍니다. 그러므로 *out* 과 *inout* 모디파이어는 캐릭터 포인터에 대해서 이렇다하게 특별하게 처리해 주질 않습니다. 그러므로 스트링을 pass-by-reference 로 전달하려면 한단계 더 간접적인 방식을 취해야 합니다.

```
- getTuneTitle: (out char **)theTitle;
```

객체 역시 마찬가지입니다.

```
- adjustRectangle: ( inout Rectangle **)theRect;
```

이런 사항은 runtime 이 수행하는 것이지 컴파일러가 하는 것이 아닙니다.

NOTE : 좀 이상하긴 합니다. 원래 *를 이용한 케이스를 봤을때는, 이해가 갑니다. 즉 스트링을 보내려하면, runtime 은 스트링을 가르키는 포인터를 보고, 그 내용을 까 본 후, 그 내용을 보냅니다. 즉 스트링이 “this”이면 이것을 가르키는 포인터는 사실 ‘t’만 가르키기에 ‘t’만 보내겠죠. 그렇다면 **을 이용한 케이스를 보면 이때는 그 스트링을 가르키는 어드레스를 보낼 것입니다. 왜냐하면 한번 dereference 를 해서 보내는 메커니즘이기에, 한번 dereference 를 하면 나오는 것은 그 스트링에 대한 pointer 이기 때문입니다. 문제는 이 pointer 가 가르키는 공간이 sender 측이 주소 공간이라는 것입니다. 그렇다면 수신측에서는 받은 내용 자체가 송신측의 주소공간이란 소립니다. 이것은 앞에서 살펴본 송신측 주소공간을 수신측의 것으로 바꾸는 것과는 다릅니다. 여기서는 바로 그 주소 자체가 value 이기 때문입니다. 아마도 runtime 이 이것을 적절하게 translation 하는 기능을 가진 것으로 보입니다. 즉 한번 리퍼런스를 한것이 여전히 포인터인지 아닌지를 구별해서 처리해 줄거라는 뜻입니다.

13.2.3 프록시와 카피본 (Proxy and Copies)

자 마지막으로 객체를 인자로 넘길때를 살펴봅시다.

```
- danceWith: ( id )aPartner;
```

이 메소드는 aPartner 라는 객체에 대한 id 를 수신측에 보냅니다. 송신측과 수신측이 같은 프로그램에 속한다면 같은 aPartner 객체를 가르키고 있을겁니다.

수신측이 원격 프로그램이면 어떨까요? 역시 마찬가지로 같은 객체를 가르키고 있습니다. 하지만 단 예외가 있습니다. 수신자가 그 객체를 proxy 를 통해서 접근하고자 한다면, (왜냐하면 그 객체가 수신측의 주소 공간에 있지 않으니까) 같은 객체를 가르키고 있지 않게 됩니다. 수신측이 받는 것은 proxy 에 대한 id 이지 원래의 객체에 대한 id 가 아닙니다. proxy 에 보내진 메시지는 물론 실제의 객체로 전달은 되고, 반환 역시 마찬가지입니다.

그런데 proxy 를 사용하는게 불필요하게 비효율적일때가 있습니다. 이럴때는 차라리 그 객체의 카피본을 원격 프로세스에 직접 보내고, 수신측에선 직접 그 객체를 자신의 주소 공간에서 억세스하는 편이 낫습니다. 이렇게 하기위해서 Objective-C 는 *bycopy* 라는 모디파이어를 제공하고 있습니다.

```
- danceWith: ( bycopy id )aClone;
```

이 bycopy 는 값을 반환하는데도 쓸 수 있습니다.

```
- ( bycopy)dancer;
```

pass-by-reference 를 이용하는 경우, 즉 pointer 를 이용하는 경우는 *out* 과 함께 쓸 수 있습니다.

```
- getDancer: ( bycopy out id *)theDancer;
```

NOTE : 객체의 카피본이 다른 프로그램으로 전달되었을 경우엔, 그것이 익명이어서는 안됩니다. 즉 객체를 받는 원격 프로그램은 해당 객체에 대한 클래스를 자신의 주소공간에 로딩해야 합니다.

bycopy 는 어떤 클래스들에 사용하면 상당히 의미가 있겠습니다. 즉 어떤 클래스가 여러 다른 객체를 포함한다고 합시다. 이런 경우에 *bycopy* 를 사용하면 원격 프로세스가 해당 서브 객체들 바로 억세스할 수가 있겠습니다. 물론 *byref* 를 이용하면 이것을 override 할 수 있습니다. 하지만 그렇게 하면 반드시 reference 를 이용해서 전달되거나 반환되어야 합니다. 대부분의 Objective-C 객체들에선 pass-by-reference 가 기본이기에, 아마 *byref* 를 쓸 일은 거의 없을겁니다.

이와 같이 변경을 가하기 위해서 bycopy 나 byref 를 이용하기에 적당한 단 하나의 타입은 객체일 경우뿐입니다. 그게 동적으로 된 id 이거나 정적으로 클래스 이름으로 typing 이 되었건 말입니다.

앞에서도 언급되었다시피, bycopy 나 byref 는 클래스나 카테고리 선언에는 사용될 수 없지만, formal protocol 내부에선 사용될 수 있습니다. 다음의 예를 봅시다.

```
@Protocol foo
    - (bycopy)array;
@end
```

이제 클래스나 카테고리가 위의 프로토콜을 채용할 수있습니다. 이렇게 하면 객체들이 어떻게 전달되고 반환될 수 있는지 힌트를 주게끔 프로토콜을 만들 수 있습니다.

Chapter 14. Type Encoding

runtime 을 보조해주기 위해서, 컴파일러는 return type 이나 인자 type 들을 캐릭터 스트링으로 인코딩해서, 해당 메소드의 selector 들과 연결을 시켜줍니다. 이런 것은 다른 용도로도 쓸모가 있기 때문에 *@encode()*라는 컴파일러 디렉티브를 이용해서 쓸 수 있도록 공개했습니다.

즉 타입을 주면, *@encode()*는 그 타입에 맞는 스트링 인코딩을 반환해 줍니다. 이런 타입으로는 *int* 같이 primitive 한 타입일 수도 있고, pointer 나 tag 가 된 structure 나 union, 혹은 클래스 이름이 될 수있겠습니다. 좀더 명확히 말하자면 C 의 sizeof() 연산자의 인자로 쓰일 수 있는 것이면 아무거나 다 됩니다.

```
char *buf1 = @encode(int **);
char *buf2 = @encode(struct key);
char *buf3 = @encode(Rectangle);
```

밑의 테이블에 타입 코드들을 뽑아 놓았습니다. 저장(archiving)이나 배포를 목적으로 객체를 인코딩할 때쓰는 코드와 일치하는게 많다는 것을 염두에 두십시오. 하지만 coder 를 작성할때는 사용할 수 없는 코드도 있으며, @encoder()가 만들지는 않지만, coder 를 만들때 쓰고 싶은게 있을 수도 있을겁니다. 여기에 대해서는 Founcation framework 리퍼런스의 NSCoder 클래스 스펙을 살펴보십시오.

Code	Meaning
c	char
i	int
s	short
l	long
q	long long
C	unsigned char
I	unsigned int
S	unsigned short
L	unsigned long
Q	unsigned long long
f	float
d	double
B	C++ bool or C99 _Bool

Code	Meaning
v	void
*	character string (char *)
@	an object (whether it is static or dynamic id)
#	class object (Class)

:	method selector (SEL)
[array type]	array
{name=type…}	structure
(type…)	union
bnum	bit field of num bits
^type	pointer to type
?	unknown type (function pointer 용으로 쓰임)

이 맵을 이용해서 어떻게 타입이 coding 되는지 알아보자.

float t[12];

이와 같은 경우는 우선 배열이므로 대괄호(bracket)으로 둘러싸이게 된다. 그리고 포인터이므로 ^이 쓰이게 된다. 이것은 결과적으로 다음과 같이 바뀌게 된다.

[12^f]

구조체는 중괄호(brace)인 {,}에 둘러 쌓인다. 유니온은 소괄호(parenthes)인 (,)에 둘러 쌓이게 된다. structure 의 tag 이 먼저 나오고, 그다음은 = 이, 그 다음은 그 구조체 안에 정의된 필드들에 대한 코드가 일렬로 나오게 된다. 다음의 예를 보자.

```
typedef struct example {
        id anObject;
        char *aString;
        int anInt;
} Example;
```

이 경우는 다음과 같이 변하게 된다.

```
{example=@*i}
```

이때 타입의 이름인 Example 을 쓰던, 구조체의 tag 인 example 을 쓰던 결과는 같다.
그리고 이 구조체에 대한 포인터를 encode()에 넘기면 결과는 다음과 같다.

```
^{example=@*i}
```

하지만 포인터에 포인터를 쓰면 내부 타입 정보가 더 이상 나타나지 않게 된다.

```
^^{example }
```

객체의 경우는 구조체의 경우처럼 되는데, NSObject 를 @encode()에 넘기면 다음과 같이 나온다.

^{NSObject=#}

뒤에 #이 붙는 이유는 NSObject 에는 단 한개의 인스턴스 변수가 선언되어 있는데, isa 라는 것이 그것이다. 이것의 타입이 클래스이기 때문에 #이 붙는다.

이외에도 runtime 시스템이 사용하는 것들이 있는데, 이는 @encode()가 결과를 반환해주지는 않는것이다. 이것들은 프로토콜에 선언된 메소드를 위해서 존재한다.

Code	Meaning
r	const
n	in
N	inout
o	out
O	bycopy
R	byref
V	oneway

맺는말

부족하나마 이렇게 Apple 의 Objective-C 에 대한 입문서를 만들었습니다. 앞에서 언급했듯이 Apple 의 문서를 보고하다보니 시간이 지날 수록 번역본이 되어갔다는 점, 독자들이 읽으면서 따라하면 Objective-C 프로그래밍에 익숙해지도록 하는 많은 예제가 없다는 점이 안타깝지만 우선 한글판 Objective-C 설명서를 하나 내자는 마음과, 게으름 때문에 이 정도에서 마무리하게 되었습니다.

Objective-C 는 다른 언어와 달리 Cocoa 와 따로 떼어서 생각할 수 없습니다. 왜냐하면 많은 유용한 것들이 언어자체에서 지원을 해주는 것 외에도 Cocoa 프레임 워크를 통해서 지원되기 때문입니다. Objective-C 판 Standard C 라이브러리의 위치를 차지하는 Core Foundation 은 더 말할 나위 없겠습니다. 즉 언어자체는 최소화/간략화를 염두에 두고 발전되어 왔다는 것을 알 수 있습니다.

하지만 얼마전에 Apple 에서 CoreFoundation Lite 를 오픈 소스화해서 놓았으므로, C 와 Standard C 라이브러리, 혹은 C++과 Standard C++ 라이브러리의 모양을 어느 정도 갖추었다고도 할 수 있겠습니다. 바야흐로 Linux 나 Windows 에서도 Objective-C 프로그래밍을 해 볼 수있게 된 것입니다. 그러므로 Objective-C 의 기능이 C++에 비해서 미약해 보일 수는 있지만 이런 라이브러리와 같이 쓴다면 결코 그렇지 않음을 아시게 되실거라고 봅니다.

요새는 C++에서 dynamic 한 요소를 많이 첨가하여 Objective-C 나름의 장점은 많이 없어진게 아닌가 싶습니다. 하지만 garbage collection 이 되며, 굉장히 단순한 C 의 확장으로 복잡한 C++이 하는 많은 것을 한다는 점은 상당히 고무적인 일이라고 생각합니다. 개인적으로 Objective-C 가 많이 쓰였으면 좋겠지만, 그렇다고 확장에만 중점을 두어서 지저분한 언어가 되는 것은 바라지 않습니다.

이렇게 Objective-C 에 대해서 알아보았으면, 그 다음은 어떤 것을 읽어야 할까요? MacOS X 에서 실제 GUI 프로그래밍을 위한 갖가지 클래스들을 알아보는 것도 중요하지만, 그보다 먼저, MacOS X 의 event loop 등에 대한 것을 알아야 합니다. 그래야 Interface Builder 에서 연결시키는 것에 대한 이해가 가능하리라고 봅니다.

개인적인 욕심으론 그것들을 모두 한글화하고 싶지만, 당분간 여기서 접으려 합니다.

이젠 저 스스로도 프로그래밍을 해야 하고, Visual C++/C# .NET 도 공부해야 하겠다는 절박감이 들어서 입니다. 또한 Finance 쪽도 공부를 해야하니 시간이 정말 없습니다. 안타까운점은 좋아하는 플랫폼에서 시간을 많이 보내고 싶지만, 그러질 못해 실력이 늘지 않는다는데 있습니다.

요번 WWDC 2005 에서 보고 느낀 점은, 의외로 이 작은 시장에서 활동하는 사람들이 대단히 많다는 것이며, 그 숫자는 물론 Windows 의 개발자에 비하면 적은 숫자지만, 그렇다고 해서 절대적으로 적은 숫자도 아니라는 점입니다. Saturation 이 된 시장에서의 개발과 재미있는 요소가 많이 남아 있는 시장에서의 활동.. 세상엔 참 능동적으로 살아가는 사람들이 많다는 것을 보았습니다.

모쪼록 이 책이 남들과 달리 생각하시고 살아가시는 분들에게 조금이나마 도움이 되었으면 합니다.

이 도큐먼트에 수정할 내용을 발견하시거나 더할 내용이 있거나하면 언제든지 연락해 주시기 바랍니다.

감사합니다.

2005 년 6 월 19 일 타향에서..
박종암 씀

jongampark@gmail.com
861 South Catalina Street #309
Los Angeles CA 90005