GNU Smalltalk 로 배우는

컴퓨터 프로그래밍

Canol Gökels

번역 : 조성재 (cho.sungjae@gmail.com)
감수 : 오화종 (hwajongoh@me.com)
김승범 (picxenk@gmail.com)

한국 스몰토크 사용자 그룹 (Korean Smalltalk User Group)

번역 동기는 불순하였지만,
번역을 부추긴 장본인의 열정에 등떠밀려
이 책을 번역하였습니다.
스몰토크에 몰두하는 양파옹에게 이 번역물을 바칩니다.
- 역자 조성재

Last updated: 18.10.2009

한국어 번역일 : 2011 년 10 월 31 일

한국어 마지막 갱신일 : 2012 년 1 월 24 일

Cover photo by: Tibor Fazakas over www.sxc.hu

어제 실수로 밟은 개미에게 바칩니다...

Canol Gökel

서문

컴퓨터는 하드웨어와 소프트웨어로 구성된 기계입니다. 소프트웨어는 하드웨어를 위해 작성된 것이고요. 여러분이 단순히 인터넷만 사용하는 컴퓨터 사용자이거나, 매일하는 작업을 위해 컴퓨터를 사용하는 사람이거나에 관계없이 특정 목적을 위해 프로그래밍 해야 할 때가 있을 것입니다. 여러분은 아마, 컴퓨터가 여러분을 지배하기 전에, 여러분의 컴퓨터를 지배하길 원하는, 호기심 많은 사용자일 것입니다. 프로그래밍은 컴퓨터를 지배하기에서 중요한 부분입니다. 프로그래밍은 몸에 영혼을 불어넣는 것과 같은 작업이니까요.

이 책은 GNU 스몰토크(Smalltalk) 프로그래밍 언어를 사용하여 프로그래밍의 기본을 가르쳐주는 책입니다. GNU 스몰토크는 Smalltalk-80 프로그래밍 언어의 구현이며, 일반적으로 사용하는 C 나 Java 프로그래밍 언어와는 약간 다른, 스몰토크 언어군의 언어입니다. 그래서 이 책에선 다른 책들과 다른 방법으로 프로그래밍에 접근하도록 하겠습니다.

이 책의 챕터들이 대부분 다른 프로그래밍 책들에 비해서 짧다는 것을 알게 될 것입니다. 이유는 두 가지인데, 하나는 저의 게으름 때문이고, 하나는 스몰토크가 작고 군더더기가 없는 언어이기 때문입니다. 작기 때문에, 다른 언어에 비해 여러분이 배워야 하는 프로그래밍 개념이 적습니다. 스몰토크는 설계와 개념 수립에 아주 세심하게 신경을 써서 만들었습니다. 군더더기가 없기 때문에, 다른 언어에 비해 예외가 적은 편입니다. 이 두 가지 큰 이유 덕분에, 여러분이 100 페이지 내의 내용만으로도 이 언어 전체를 익힐 수 있습니다.

위의 내용은 스몰토크로 여러분이 할 수 있는 것을 제한한다는 이야기가 아닙니다. 대조적으로 작은 언어요소 집합과 모순점이 없다는 것이 여러분에게 무한한 유연성을 제공합니다. 오로지 여러분의 상상력만이 한계가 될 뿐이지요. 또한, 스몰토크 최고의 장점 중 하나인 기본 탑재 라이브러리는 당신이 필요한 대부분의 도구를 제공해 줄 것입니다. GNU 스몰토크는 (표준 스몰토크가 제공하는) 도구에 더 많은 도구를 제공합니다. 하지만, 이 책이 첫 번째 판이고, 스몰토크의 팁을 알려주기 위한 책이기 때문에, 가장 중요하고, 자주 쓰이는 라이브러리의 기능만 기술하고, 스몰토크의 핵심 언어에 대해서만 집중하여 다룰 것입니다.

만약 여러분이 스몰토크를 배우고자 하는, 경험이 풍부한 프로그래머라면, 이렇게 섬세하고 꼼꼼하게 신경써서 만든 언어의 정밀함에 깜짝 놀라게 될 것입니다. Andrew S. Tanenbaum 은 "다양한 기능을 넣으려 하지 마라. 소프트웨어를 안전하고, 신뢰성 있고, 빠르게 만드는 방법은 작게 만드는 것 뿐이다." 라고 말했습니다. 과학자들은 이 격언을 마음에 새겨서 스몰토크를 설계하였습니다.

이 책은 누구를 위한 것입니까?

이 책은 컴퓨터를 사용할 줄은 알지만, 컴퓨터 프로그래밍에 대한 경험이 없는 독자들을 대상으로 설명하고 있습니다. 컴퓨터를 사용할 줄 안다는 의미는 어떻게 파일을 열고, 닫고, 프로그램을 실행하고, 저장할 수 있는지 등의 경험을 가지고 있는 것을 말합니

다.

> 프로그래밍 유경험자들에게:
>
> 우리는 이 상자글을 통해, 다른 프로그래밍 언어 지식이 있는 독자들에게 무언가를 말할 것입니다. 처음 프로그래밍을 접한 독자들은 읽을 필요는 없습니다.

> 프로그래밍 유경험자들에게:
>
> 이 책은 프로그래밍을 경험한 독자들을 위해서도 사용할 수 있습니다. 실제로 이 책은 여러분이 경험했던 프로그래밍 언어와 상당히 다르기 때문에, 처음 스몰토크를 접한 분들께서는 꼭 읽어야 할 책입니다.

이 책은 어떻게 사용할까요?

만약 여러분이 정말 아무것도 모르는 초보라면, 처음부터 차근차근 순서대로 읽으시길 권합니다. 그리고 다른 곳에서 참조한 내용의 경우, 뒷 편의 색인을 보고 찾아보시길 바랍니다. 편찬을 할 때, 순서대로 글을 읽을 수 있도록 쓰려고 노력했습니다. 모르는 개념이 나오는 부분을 일부러 미리 볼 필요가 없도록 하기 위해서 입니다. 부록 부분을 보기 위해 엄청 많은 페이지를 앞질러 봐야 할 때도 있겠지만, 본문과 관련하여서는 말한 대로 순서대로 읽으시면 됩니다. 그리고 예제는 가능한 단순하게 만들었습니다. 불필요한 세부사항 때문에 혼동을 가져오지 않도록 하기 위한 것이며, 동시에 다루고 있는 개념에 대해서 만 설명하기 위한 조치입니다.

프로그래밍 언어를 배우는 동안에, 이론으로 배운 내용을 연습하는 것은 상당히 중요합니다. 그래서 새로운 개념을 소개한 후에는 많은 예제를 제공하려고 노력했습니다. 코드를 만들려고 하고 혼자서 예제 프로그램을 실행시켜보는 것은 개념을 이해했다는 것을 확인하는 것일 뿐만 아니라, 자신감과 공부를 지속할 수 있는 동기를 줄 것입니다. 또한 각 장에서 프로그래밍 해결 방법을 생각하고 (아마도 적용할 수도 있는) 작은 질문도 넣어 두었습니다. 마지막으로 각 장을 읽고 난 후, 새로운 개념에 대한 여러분의 지식을 더욱 강화할 수 있는 연습문제도 준비해두었습니다. 이 문제들을 꼭 풀어보시길 권장합니다. 답안은 이 책의 부록 B 에서 확인할 수 있습니다.

또한 대부분의 프로그램들은 다음 온라인 주소에서 다운로드 받아 사용할 수 있습니다.

http://www.canol.info/books/computer_programming_using_gnu_smalltalk/source_code.zip

글꼴별 전달 정보

본문 내용 전체에서, 글꼴의 모양을 다르게 하여 설명하는 내용 중 구별해야 하는 부분을 알 수 있도록 글꼴을 사용하였습니다.

일반 본문에는 나눔 명조체를 사용하였습니다. 글꼴 모양이 *기울어진* 부분은 책에 처음 나왔거나, 정의할 단어를 강조하기 위한 표시입니다.

코드 부분을 다룰 때에는 고정폭 (나눔고딕코딩 병행) 글꼴을 사용하였습니다.

여러분의 취향, 프로그램의 문맥, 여러분의 컴퓨터의 설정에 따라 달라져야 하는 코드에는 *기울어진 고정폭* 글꼴을 사용합니다.

다루고 있는 소스코드 중에서 의미있는 부분은, 아래와 같이 연보라빛 사각형을 표시해두었습니다.

```
a stand-alone meaningful code
```

프로그램의 실행 결과를 표시할 때에는 다음과 같이 보라빛 배경에 표시합니다.

```
Output of a program
```

프로그램을 실행하는 동안 입력부분은 출력 부분에서 두꺼운 글꼴로 표시합니다.

```
Please enter your name: Canol
```

키보드 상의 키를 언급할 때에는, <Ctrl>, <Enter>, <Backspace>[1]와 같이 꺾인 괄호 부호 사이에 키 이름을 씁니다.

또한 몇몇 특별한 상자도 있습니다.

> 주의:
>
> 이 상자는 중요한 세부사항, 추가적인 정보와 제안들을 담고 있습니다.

> 프로그래밍 유경험자들에게:
>
> 이 상자는 다른 프로그래밍 언어를 알고 있지만 스몰토크를 배우기 위해 이 책을 읽는 사람들에게 주는 힌트를 포함하고 있습니다. 초심자나 경험있는 프로그래머들도 이 상자의 내용을 꼭 이해하거나 읽을 필요는 없습니다.

> 질문:
>
> 이 상자는 몇몇 특별한 질문을 담고 있습니다. 각 장의 끝 부분에 답이 있습니다.

1 이 책에서는 PC 의 <Enter> 키와 맥킨토시의 <Return> 키를 모두 <Enter>키로 통일해서 표기하겠습니다.

목차

스타트렉의 컴퓨터는 별로 좋아보이지는 않는다. 승무원들이 아무
질문이나 하면, 컴퓨터는 한동안 생각한다.
내 생각엔 우리는 그것보다 더 나은 것을 만들 수 있다.

레리 페이지 (Larry Page)

제 1 장 프로그램 세계에 대한 소개

이 장에서, 우리는 프로그래밍 세계로 뛰어들 것입니다. 프로그래밍이란 무엇인지, 프로그래밍 언어가 무엇인지, 프로그래밍을 시작하기 전에 여러분들이 익숙해져야 할 기본 주제들은 무엇인지에 대해 집중해서 보길 바랍니다.

프로그래밍이란 무엇이고, 프로그래밍 언어는 무엇입니까?

프로그래밍 *programming* 은 우리가 원하는 무엇인가를 컴퓨터가 하도록 만드는 것입니다. 그리고 *프로그램* *program* (혹은 *소프트웨어* *software*) 은 하나의 특정 작업을 완수할 수 있는, 컴퓨터 코드의 묶음에 붙이는 이름입니다. 때때로 프로그램 제작 과정이 문제 해결 과정이라고 말하기도 합니다. 특정 문제를 해결하도록 모든 프로그램을 제작하기 때문에, 실제 여러분이 직면한 문제를 푸는 것이라고 말할 수 있습니다.

하드웨어 설계 때문에 컴퓨터는 오로지 두 가지 상태만을 이해할 수 있습니다. 우리는 이 두 가지 상태를 디지털 0 과 1 로 표현합니다. (이것에 관하여, 이 장 나중에 더 자세히 설명하도록 하겠습니다.) 0 과 1 의 조합은 컴퓨터 하드웨어가 이해할 수 있는 여러 명령이 됩니다. 컴퓨터와 인간이 소통할 수 있는 이러한 명령어 그룹을 *프로그래밍 언어* *programming languages* 라고 부릅니다.

단순히 0 과 1 로 명령어를 쓸 수 있지만, 사람에게는 무척 고통스러운 방법입니다. 0 과 1 로 가득찬 화면을 상상해보십시오. 한 눈에 그것을 명령어로 인식하기 위해서는 수년간의 경험이 필요할 것입니다. 따라서 인류는 프로그램을 작성하고 실행하는 과정 사이에 중간과정을 개발하는 것으로 이러한 어려움을 해결하였습니다. 이 중간과정은 대부분 영어와 비슷한 문법으로 이해하기 쉽게 프로그램을 작성하고 나서, 하드웨어가 이해할 수 있는 기계어 machine language 혹은 기계 코드 machine code 라 불리는 0 과 1 의 조합으로 전환하는 과정입니다. 이런 전환과정 또한, 컴파일러 *compiler* 혹은 인터프리터 interpreter 라 불리는 프로그램을 수행하여 해결합니다.

프로그래밍 언어 형식

프로그래밍 언어는 다양한 속성에 따라 분류합니다. 몇 가지 속성에 대해 이야기해보겠습니다.

고수준 그리고 저수준 프로그래밍 언어

앞에서 먼저 말한 것처럼, 기계어는 인간이 이해하기 힘듭니다. 따라서 오늘날 인간이 사용하는 언어와 비슷한 언어를 만들었습니다. 이러한 언어는 얼마나 인간의 언어와 가까운가에 따라 분류할 수 있습니다. 기계어에 가까울수록, 저수준 프로그래밍 언어라 합니다. 반대로 인간의 언어에 가까울 수록 고수준 언어라고 합니다.

다음의 기계어를 보십시오.

```
01101011101000101110000101
```

여러분은 이것이 무엇을 의미하는지 상상조차 할 수 있겠습니까? 이제 *어셈블리* *Assembly* 라고 불리는 언어로 어떻게 표현하는지 봅시다.

```
MOVE  d'3', W
ADD   d'4'
```

이 코드가 무엇인지 여러분이 추측할 수 있는지는 모르겠습니다만, 기계 코드보다는 더 읽기 쉽다는 것은 명백합니다. 어셈블리 언어는 기계어보다는 더 고수준의 언어라고 말할 수 있겠습니다. 위의 두 줄의 코드가 무엇을 하는 것인지 궁금하십니까? 이 코드는 단순히 두 숫자를 더하는 내용입니다.

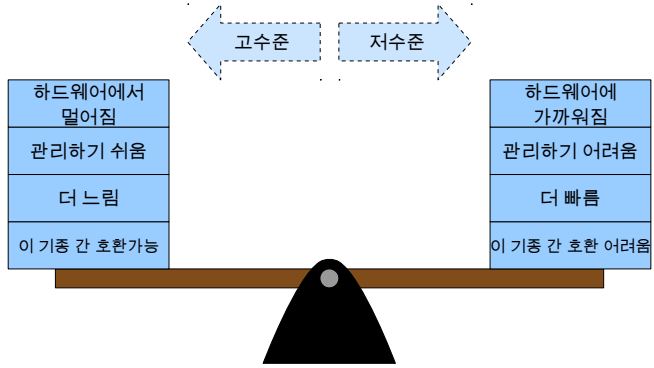
이제 GNU 스몰토크라고 불리는, 프로그래밍 언어로 쓰여진 같은 내용의 코드를 봅시다 (처음 들어보는 언어인데 말이죠).

```
3 + 4
```

이 코드가 무슨 역할을 하는지 굳이 여러분에게 물어보고 싶지는 않습니다. 만약 여러분이 모르겠다고 말씀하신다면, 이보다 더 명확하게 위의 코드가 무슨 역할을 하는지 설명할 수 있는 방법을 저는 모르겠습니다. 그러므로 기계어와 어셈블리, GNU 스몰토크를 비교해봤을 때, 가장 고수준의 언어는 GNU 스몰토크라고 할 수 있습니다.

예제가 아주 간단하고 명료해서, GNU 스몰토크가 진짜 영어같다고 기대하지는 마십시오. 최근에 자주 사용하는 다른 프로그래밍 언어와 비교하여 아주 고수준의 언어라는 평판을 가지고 있지만, 때때로 어셈블리만큼이나 어려울 수 있습니다.

의미심장한 격언으로 이번 주제를 끝마칠까 합니다. '더 고수준의 언어가 더 나은 언어를 의미하는 것은 아니다.' 오늘날 어셈블리 역시 매우 일반적으로 사용되는 언어입니다. 왜냐하면 언어가 하드웨어에 가까울수록 하드웨어를 제어하는 데에 더 강력하기 때문입니다. 따라서 근원적인 하드웨어 프로그래밍을 하려 한다면, 아마도 어셈블리가 좋은 선택이 될 것입니다. 언어의 수준이 낮을수록 최적화의 수준을 높일 수 있어서, 더 빠른 프로그램을 만들 수 있습니다. 물론, 성능향상을 위해 어셈블리로 오피스 프로그램 같은 프로그램 전체를 만들겠다는 것은 어리석은 이야기입니다. 결과가 나올 때까지, 일반 고수준 언어로 개발하는 것보다 수백 배에 달하는, 어마어마한 개발 시간과 엄청난 복잡도, 읽을 수 없는 소스코드들을 생각해보면 왜 고수준 언어로 관리를 해야 하는지 알 수 있습니다. 따라서 여러분들은 현재 프로젝트에 대한 프로그래밍 언어를 선택하기 전에 프로젝트의 성격과 조건을 고려해봐야 합니다. 이는 다른 프로젝트에 다른 언어를 사용할 필요가 있다는 것을 의미합니다.

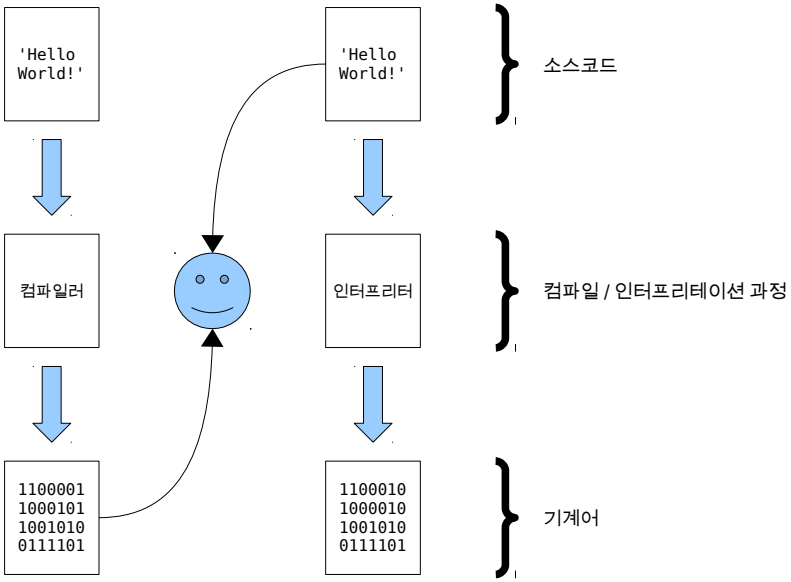


컴파일 언어, 인터프리터 언어

프로그래밍 언어로 프로그램을 작성한다는 것은 설명했습니다. 이번엔 우리가 작성한 프로그램을 사람이 읽을 수 있는 프로그램에서 컴퓨터가 이해할 수 있는 형식으로 변환하는 단계에 대해 설명하겠습니다. 만약, 전체 프로그램이 한번에 기계어로 변환되어 메모리에 올라간다면 이 과정을 *컴파일* *compilation* 이라고 합니다. 소프트웨어를 작성할 때, 실행하기 전에 컴파일 해야 하는 프로그래밍 언어로 작성되었다면, 이 언어는 *컴파일 언어* *compiled language* 로 부릅니다. 이러한 컴파일 과정을 진행하는 소프트웨어를 *컴파일러* *compiler* 라고 부릅니다.

만약 프로그램을 우리가 쓴 대로 바로 변환한다면, 즉 한 줄, 한 줄씩, 더 정확하게는 명령문 하나씩을 바로 변환한다면, 이 과정을 *인터프리테이션* *Interpretation* 이라고 하며, 그러한 언어들을 *인터프리터 언어* *interpreted language* 라고 합니다. 인터프리터 언어로 작성한 소프트웨어는 우리가 작성한 방식으로 유지하며, 매 회 실행할 때마다 기계어로 다시 번역하여 실행합니다. 이 과정을 실행하는 소프트웨어를 *인터프리터* *interpreter* 라고 부릅니다.

(컴파일 결과나 인터프리테이션 과정의 결과가 아닌) 프로그래밍 언어로 쓴 코드를 *소스코드* *source code* 라고 부릅니다.



위 그림은 컴파일 한 프로그램과 인터프리터 프로그램의 실행 과정 차이를 보여주고 있습니다. 사용자는 프로그램을 전달 받을 때, 컴파일 언어로 이용한 경우, 컴퓨터 메모리에 바로 기계어로 올릴 수 있는 프로그램을 받을 수 있지만, 인터프리터 언어로 작성한 경우, 소스코드만을 받아볼 수 있습니다. 인터프리터 언어로 만든 프로그램을 실행 할 때, 시스템은 소스코드 파일 형식을 인식하고, 기계어로 번역하기 위해 소스코드를 인터프리터로 전달합니다. 가끔은 시스템이 파일 형식을 인식하지 못하여서, 사용자가 직접 인터프리터에 파일을 전달해야 할 때가 있습니다. 다시 말해, 인터프리터 언어는 실행할 때마다 추가 단계를 거쳐야 하며, 이로 인해 실행 시간이 느려집니다.

실제로 같은 언어라도 그 구조에 따라 컴파일러와 인터프리터 모두 존재할 수 있지만, 대부분의 언어는 이 중 한 가지 언어로 설계합니다. 따라서 언어의 컴파일러 버전과 인터프리터 버전간에 차이가 있을 수 있습니다.

비록 프로그래밍 언어의 두 형태가 가장 인기있다 하더라도, 두 페러다임 외의 다른 형태의 언어 또한 있습니다. 이 페러다임은 소스코드를 인간이나 컴퓨터 모두 이해할 수 없는 중간 수준의 언어로 컴파일합니다. 이러한 수준의 변환을 보통 *바이트코드* *byte-code* 컴파일이라고 부릅니다. 이 과정은 하드웨어에 더 가까운 수준으로 소스코드를 변환하여, 하드웨어에서 사용하려 할 때, 기계어로 더 빠르게 변환할 수 있도록 만드는 것만으로 끝납니다. 바이트코드에서 기계어로 변환하는 과정은 *가상머신* *virtual machine* 이라고 불리는 프로그램이 수행합니다. 가상머신이 하는 일은 사용자가 프로그램을 실행할 때 바이트코드를 기계어로 인터프리팅하는 것입니다.

왜 이런 제 삼의 언어가 존재하느냐 하면, 인터프리터 언어가 컴파일 언어에 비해 실행면에선 느린 반면 크로스 플랫폼의 잇점이 있기 때문입니다. *크로스 플랫폼* *cross-platform* 이란 다른 OS 나 프로세서 같은, 구조가 다른 컴퓨터상에서도 소프트웨어를 실행할 수 있

다는 것을 의미합니다. 따라서 여러분이 작업하는 컴퓨터 시스템 안에 같은 인터프리터 언어가 있다면, 인터프리터 언어로 작성한 프로그램은 어떤 시스템에서나 실행할 수 있습니다. 어떤 하드웨어인지, OS 인지 상관하지 않습니다. 만약 컴파일 언어로 소프트웨어를 컴파일한다면, 컴파일 했던 시스템과 같은 환경의 프로세서, OS 에서만 실행할 수 있습니다. 컴파일 언어는 컴파일 한 후, 실행속도가 빠르다는 장점이 있지만, 소스코드에서 수정한 사항이 있을 때마다, 그리고 소프트웨어를 다른 플랫폼으로 옮길 때마다 다시 컴파일해야 한다는 단점이 있습니다. 바이트코드로 프로그램을 컴파일 한다는 것은 이러한 인터프리터 과정을 더 빠르게 만들고, 인터프리터 언어의 장점도 가지고 있다는 것을 의미합니다.

GNU 스몰토크는 좀 더 인터프리터 언어처럼 작동하기는 하지만, 스몰토크는 가상 머신을 사용하는 언어로 설계하였습니다. 이런 형태의 언어로서 (또한, 가장 성공한 언어의 예로서) Java 프로그래밍 언어가 있습니다.

절차언어, 함수형 언어, 객체 지향 언어

또 다른 프로그래밍 언어의 범주는 언어의 페러다임에 따른 것입니다. *프로그래밍 패러다임* *programming paradigm* 이란, 문제 해결을 위해 프로그래밍 언어가 문제를 어떻게 바라보냐는 것입니다. 주로 세 가지의 프로그래밍 페러다임이 유명한데, 절차언어, 함수형 언어, 객체 지향 언어가 그것입니다. 간단하게 설명하기 위해, 그리고 여러분이 아직 모르는 용어에 대한 혼동을 피하기 위해 스몰토크의 핵심인 객체 지향 언어에 대한 설명만 하기로 하겠습니다.

객체지향 프로그래밍 *object oriented programming* 은 객체들로 조합된 객체로 '세계'를 바라봅니다. 이 페러다임에 따르면 모든 것은 객체로 생각할 수 있습니다. 예를 들어, 컴퓨터, 텔레비전, 책 등 이 모든 것이 객체입니다. 또한, 이 객체들은 다른 객체들의 조합입니다. 예를들어, 컴퓨터는 메인보드, 그래픽 카드, 하드디스크, 그리고 다른 하드웨어로 구성되는데, 이 모든 것 역시 객체입니다.

객체지향 프로그래밍은 일반적으로 인간의 사고와 가장 근접한 프로그래밍 페러다임으로 여깁니다. 따라서 여러분이 객체지향 프로그램을 작성할 때에는 더욱 편안함을 느끼고, 프로그래밍 언어의 구조나 규칙 등을 걱정하는 대신 문제 해결에 집중할 수 있습니다.

프로그래밍 언어가 한가지 패러다임에만 한정되어 있을 의무는 없으며, 언어 구현 또한 그렇습니다. 실제로 개발자들이 프로그래밍을 하는 동안, 하나 이상의 패러다임으로 구현할 수 있는 언어를 *멀티 패러다임 프로그래밍 언어* *multi-paradigm programming language* 라고 합니다. 프로그래머가 한 가지 패러다임 만을 사용하여 프로그래밍 하도록 만든 언어를 *싱글 패러다임 프로그래밍 언어* *single-paradigm programming language* 라고 합니다.

스몰토크는 1 세대 객체 지향 언어 중 하나이며, 순수한 객체지향 프로그래밍 언어입니다. 즉, 싱글 패러다임 프로그래밍 언어이며, 모든 것이 다 객체입니다.

숫자 체계

이전에 컴퓨터 하드웨어가 0 과 1 만을 이해할 수 있다고 간단히 설명하였습니다. 이제는 왜 그러한지, 조금 더 자세한 이유를 설명할 것입니다. 오늘날 컴퓨터에서 일반적으로

사용하는 하드웨어는 하드웨어의 디자인 때문에 두 가지 상태만을 알아볼 수 있습니다. 예를 들어 이 두가지 상태를 on 과 off 로 부를 수 있습니다. 따라서 인간은 두 가지 기로 호만 세상을 표현할 수 있는 방법이 필요했습니다. 두 기호로 수학을 기술하는 방법은 이미 구현되어 있어서, 두 기호를 다른 것으로 나타내는 것만이 필요했습니다. 예를 들어, 컴퓨터 세계에서 냉장고를 어떻게 나타낼 수 있겠습니까? 우리는 몇몇 문자열의 집합으로 표현할 수 있습니다. 그럼 어떻게 두 가지 기호를 가지고 문자열들을 표현할 수 있겠습니까? 인간은 두 가지 기호로 문자열을 표현하는 방법을 개발했고, 이 내용은 파일 형식 주제에서 보여줄 것입니다. 몇몇 상태들을 표현할 수 있지만, 이 상태들을 변경할 필요도 있을 것입니다. 이러한 것은 몇몇 기본 명령어를 정의하고 기본 내용 위에 더욱 복잡한 것을 구축하는 것으로 달성할 수 있습니다. 이 방법은 기계어로부터 어셈블리 언어까지 그리고 우리가 이 장 이전에서 언급한 더 고수준의 언어까지 통하는 방법입니다.

> 질문:
>
> 비록 컴퓨터에서 어떤 일이 일어나는지에 대해 설명하려 했지만, 이 부분에 대한 자세한 내용은 다루지 않을 것입니다. 오로지 두 기호를 이용하여 세계를 어떻게 생성하는지 스스로 상상할 수 있을 것입니다. 이러한 것은 컴퓨터를 이해하는 데에 실제 도움이 될 것이며, 여러분들에게 흥미로운 연습이 될 것입니다. 여러분 스스로 상상해 본 뒤, 어셈블리 언어가 이 문제를 어떻게 해결했는지 보시기 바랍니다. 물론 초보자들에게 아주 어려울 것이기 때문에, 이 장의 글을 다 읽고 난 후에 보길 권장합니다.

이제 우리는 두 기호로 이루어진 세계에서의 수학적 부분에 대해 얘기할 것입니다. 비록 자세히 설명한다 하더라도, 수학 개념으로 여러분을 지루하게 만들지는 않을 것입니다. 앞으로 나올 내용을 깊이 이해하기 위해, 미리 대수학 책을 보는 것도 나쁘지 않습니다.

십 진수 숫자 체계

우리는 매일 실생활에서 (0 부터 9 까지의) 10 개의 기호로 표현하는 숫자들을 사용합니다. 왜 9 개나 11 개가 아닌 10 개였을까요? 숫자를 표현하는 데 10 개의 기호를 사용하도록 세상이 창조되었을까요? 물론 아닙니다. 우리는 우리가 원하는 만큼의 많은 기호로 숫자를 표현할 수 있습니다. 10 개의 기호를 사용하는 이유는 그것이 친숙하기 때문입니다. 최초의 인도인들이 사용한 방식이 세계로 전파되어서 그렇습니다. 왜 인도인들은 10 개의 기호로 숫자를 표현했을까요? 사람의 손가락이 10 개이기 때문이라 생각됩니다.

2 진수 숫자 체계

수학에선 우리가 숫자를 표현할 때, 원하는 만큼의 기호를 이용하여 표현할 수 있도록 허용합니다. 그러나 컴퓨터에서는 구조상 두 가지 기호만을 사용하기 때문에, 컴퓨터의 작업을 위해서는 2 진수 체계가 적합한 것입니다. 2 진수는 보통 *바이너리 숫자* *binary number* 라고 불립니다. 보통 두 기호를 0 과 1 로 표현하지만, 우리가 원하는 다른 기호를 선택할 수도 있습니다.

일단 x 진수의 숫자(여기서 x 는 자연수)를 어떻게 십진수로 변환할 수 있는지를 보여드리겠습니다. x 진수로 숫자를 쓸 때, 여러분은 각 자리의 해당 자릿값을 쓰고, 진수만큼의 배에 해당하는 경우, 윗 자리에 자릿수를 씁니다. 표현은 혼동되지만, 맞죠? 예를 들어

봅시다. 10 진수로 345 를 예로 들어보지요. 이 값은 다음과 같이 구할 수 있습니다.

$$345 = 5 \cdot 10^0 + 4 \cdot 10^1 + 3 \cdot 10^2$$

따라서 2 진수 값의 같은 수인 101011001 에 대해서도 다음과 같이 쓸 수 있습니다.

$$101011001 = 1 \cdot 2^0 + 0 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 + 1 \cdot 2^4 + 0 \cdot 2^5 + 1 \cdot 2^6 + 0 \cdot 2^7 + 1 \cdot 2^8$$

이것은 어떻게 x 진수를 10 진수로 변환할 수 있는지 보여주는 예입니다. 이제 10 진수의 수를 어떻게 2 진수로 변환하는지를 보여드리겠습니다. 345 를 2 진수로 표현하려 한다고 가정해봅시다. 우리가 할 일은 345 를 2 진수로 표현할 수 있을 때까지 계속 나누는 것입니다. 그 다음, 최종결과값과 나눗셈에서 나온 나머지 값을 오른쪽에서부터 왼쪽으로 하나씩 써두는 것입니다. 이 과정을 보여드리겠습니다.

$$345/2 = 172 \quad (remainder\ \mathbf{1})$$

$$172/2 = 86 \quad (remainder\ \mathbf{0})$$

$$86/2 = 43 \quad (remainder\ \mathbf{0})$$

$$43/2 = 21 \quad (remainder\ \mathbf{1})$$

$$21/2 = 10 \quad (remainder\ \mathbf{1})$$

$$10/2 = 5 \quad (remainder\ \mathbf{0})$$

$$5/2 = 2 \quad (remainder\ \mathbf{1})$$

$$2/2 = \mathbf{1} \quad (remainder\ \mathbf{0})$$

마지막 나눗셈 결과 값과, 나눗셈의 나머지 값을 (방정식에서 굵게 표시된 숫자들을) 차례대로 오른쪽에서부터 왼쪽으로 써 나가면, 101011001 이라는 숫자를 얻을 수 있으며, 처음 가정했던 수의 2 진수 값과 같습니다.

컴퓨터 용어에선 2 진수 한 자리를 *비트 bit* 라고 하며, 8 개의 비트를 *바이트 byte* 라고 부릅니다.

8 진수 숫자 체계

때때로 우리는 2 진수 대신, 8 진수로 표현한 숫자들을 쓸 때가 있습니다. 이유는 8 진수가 더 짧게 표현되며, 8 진수와 2 진수의 변환이 쉽기 때문입니다. 8 진수는 역사적인 사연이 있기 때문에, 배우는 것은 영광스러운 일입니다.

8 진수를 표현하기 위해, 0 부터 7 까지의 숫자를 사용할 것입니다. 이진수의 숫자를 8 진수로 변환해야 할 텐데, 이진수 오른쪽 끝부터 세 자리씩 묶어서, 각 묶음의 값을 8 진수의 값으로 변환하면 변환이 끝납니다. 예를 들어, 바이너리로 쓰여진 101011001 값을 8 진수로 표현하는 것을 보면 다음과 같습니다.

$$\underbrace{101}_{5}\,\underbrace{011}_{3}\,\underbrace{001}_{1} = 531$$

만약 숫자가 세 개 씩 나뉘어 떨어지지 않는다면, 앞에 0 을 추가하여 남은 자릿수를 채

워 넣으면 됩니다.

역으로, 8 진수의 수를 바이너리로 변환하는 것 또한 바로 가능합니다. 여러분은 8 진수의 각 자릿수 값을 2 진수로 변환하여 바로 붙이면 됩니다. 예를 들어, 8 진수 531 은 2 진수로 다음과 같이 표현합니다.

$$\underbrace{5}_{101}\ \underbrace{3}_{011}\ \underbrace{1}_{001} = 101011001$$

> 질문:
>
> 왜 이러한 변환이 항상 옳은지 설명할 수 있습니까?

16 진수 숫자 체계

이제 처음으로, 보통은 상상하기 힘든 진수, 16 진수에 대해 알아볼 것입니다. 16 진수란 숫자를 쓰는데 16 개의 기호를 쓴다는 것을 의미합니다. 그러나 아라비아 숫자 기호는 10 개 밖에 없습니다. 그럼 어떻게 해야 할까요? 수학에선 10 개 외의 나머지 기호를 쓸 수 있도록 허용되어 있습니다. 보통 라틴 알파벳 문자, A, B, C, D, E, F 를 따와서 각각 10, 11, 12, 13, 14, 15 의 값으로 사용합니다.

이 경우 우리가 "12 개의 초콜릿이 내 앞에 있다."고 표현할 때, 평소에 16 진수를 사용하는 행성에서 온 외계인은 "C 개의 초콜릿"이 있다고 말할 것입니다.

2 진수에서 16 진수로 변환을 하는 것은 2 진수 네 자리를 기준으로 하는 것을 제외하면 2 진수와 8 진수 사이를 변환하는 것과 매우 흡사합니다.

파일 형식

파일 형식 *file format* 은 컴퓨터의 메모리에 어떻게 파일을 유지하는가와 관련있는 내용입니다. 파일들은 바이너리와 텍스트 형식, 두 가지로 구분됩니다. 프로그래밍 연습을 하는 동안에는 파일들을 넓게 사용할 터인데, 실제로 모든 프로그램이 파일로 구성되어 있기 때문에 우리가 컴퓨터를 사용하는 매 시간마다, 파일을 사용하고 있는 셈입니다. 따라서 파일에 관하여 약간만 알면 쉽게 알 수 있을 것입니다.

바이너리 파일

바이너리 파일 *binary file* 은 몇몇 특별한 프로그램에서 특별한 파일 형식으로 파일을 읽기 위해 사용합니다. 컴퓨터 메모리에서 0 과 1 로 표현되며, 그 내용은 지정한 내용에 따라 생성합니다. .jpeg, .pdf, .doc, .exe 확장자들은 바이너리 파일의 예이며, 이러한 파일들을 읽기 위해서는 이미지 뷰어나, PDF 리더와 같은 프로그램이 필요합니다.

텍스트 파일들

인간은 문자로만 이뤄진 파일을 널리 사용하기에 간단한 텍스트 파일을 만들었습니다. 알파벳 문자와 숫자 및 몇몇 특수제어 문자를 포함한 텍스트 파일은 모두 *캐릭터 인코딩* *character encoding* 이라고 부르는 특수 인코딩 시스템에 따라 표시합니다. 텍스트 파일에 포함할 수 있는 문자는 캐릭터 인코딩이 사용할 수 있는 범위로 제한되어 있습니다. 이렇게 제한된 문자의 종류를 표로 나타낸 것을 *캐릭터 셋* *character set* 이라고 부릅니다. 모든

문자는 캐릭터 셋의 숫자로 표현합니다. *텍스트 에디터* *text editor* 라 불리는 프로그램으로 텍스트 파일을 읽고 쓸 수 있습니다.

일반적으로 많이 사용하는 캐릭터 인코딩은 ASCII (American Standard Code for Information Interchange, 정보 교환을 위한 북미 표준 코드) 이며, 문장에 사용할 수 있는 최소한의 문자들로, 모든 문자는 7 비트로 표현되는 간단한 인코딩 방식입니다. ASCII 캐릭터 인코딩을 지원하는 텍스트 에디터는 ASCII 로 인코딩 된 어떠한 텍스트 파일이라도 읽을 수 있습니다. 이 인코딩 방식은 캐릭터 인코딩 사양 중 가장 단순하여서 디지털 장비 간의 광범위한 호환성을 제공하고 있습니다. ASCII 캐릭터 셋은 부록 B 에서 찾아 보실 수 있습니다.

ASCII 는 7 비트의 제한을 가지고 있기 때문에 7 비트로 표현할 수 있는 128 종류의 문자 만을 사용할 수 있습니다. 세계는 크고, 수백 개의 알파벳과 수천 개의 다른 문자들이 존재하고 있습니다. 어떤 언어 체계에서는 128 개보다 더 많은 문자를 가질 수도 있기 때문에, 7 비트 인코딩으로 표현한다는 것은 불가능한 일입니다. 이런 이유로 다양한 캐릭터 셋과 인코딩이 존재하는 것입니다. 예를 들어, 유니코드 문자 셋을 사용하고 있는 UTF-8 문자 인코딩은 문자를 표현하는 데에 1~4 바이트를 사용할 수 있습니다. 유니코드 문자 집합은 약 십만 개의 문자들로 이루어져 있습니다.

GNU 스몰토크는 ASCII 나 UTF-8 로 텍스트 파일을 읽을 수 있기 때문에, 여러분의 편집기가 ASCII 나 UTF-8 을 지원하는지 여부가 문제가 되지 않습니다.

워드프로세서, 텍스트 에디터, IDE (통합 개발 환경)

우리는 소스코드를 텍스트 파일로 기록할 것이며, 문서 작성 프로그램을 소스코드 작성에 사용할 것입니다.

타이핑을 할 때에는 보통 마이크로소프트 워드나 iWorks Pages, OpenOffice.org Writer 와 같은 *워드프로세서* *word processor* 를 생각하기 마련이지만, 워드프로세서는 단순한 텍스트 파일을 만들기 위한 것은 아닙니다. 워드프로세서는 문서 내의 글꼴 굵기, 크기, 색상, 이미지, 레이아웃 등과 같은 문서 양식을 유지하기 위한 바이너리 파일을 사용합니다. 아마도 단순 텍스트 파일을 생성하기 위한 옵션을 가지고 있겠지만, 텍스트 에디터 만큼의 속도나 유용성에서의 이익을 얻을 수는 없을 것입니다. 이것은 마치 논 가는데 리무진을 타는 것과 같습니다.

텍스트 에디터 *text editor* 는 프로그래머들에게 사용하기 편하고, 빠른 해결책입니다. 여기엔 특별히 프로그래머들을 위한 몇몇 텍스트 에디터들이 있습니다. 이 에디터들은 프로그래머의 삶을 편하게 만들어 주는 유용한 요소들을 가지고 있습니다.

따라서 우리는 텍스트 파일을 만들고 관리하는 수단으로 텍스트 에디터를 사용할 것입니다. 대부분의 텍스트 에디터들은 다른 캐릭터 셋과 인코딩을 지원할 수 있도록 특화되어 있습니다. 리눅스 상에서, Gnome 환경이면 Gedit, Xfce 상에서는 mousepad, KDE 환경이면 Kate 와 같은 프로그램을 사용할 수 있습니다. 또한 Vim 이나 Emacs 같은 진보된 텍스트 에디터들도 있습니다. 윈도우즈에서도 메모장을 쓸 수 있지만, 메모장은 그리

뛰어나지 못해서 Notepad2[2]나 EditPad Lite[3] 등을 다운로드 하여 쓰길 권합니다.

구글이나 야후와 같은 검색엔진을 사용하여서 "text editor" 라고 키워드를 입력하면, 항상 좋은 결과를 얻을 수 있을 것입니다.

프로그래머는 보통 코딩, 컴파일, 프로그램 테스트를 하는 동안 다른 여러 프로그램을 필요로 합니다. 여러 프로그램을 따로따로 설치하고 사용하는 대신, 여러분이 필요로 하는 모든 프로그램을 포함하고 있는 통합 개발 환경 (IDE, Integrated Development Environment) 프로그램을 사용할 수 있습니다. IDE 로 잘 알려진 것들 중, 리눅스에선 Anjuta 나 KDevelop, 윈도우즈에선 마이크로소프트 비주얼 스튜디오나 이클립스, 코모도 같은 크로스 플랫폼 IDE 개발 환경도 있습니다. 우리는 그러한 환경을 사용하지는 않을 것입니다. 왜냐하면, 프로그래밍을 배우는 처음 단계에서는 불필요하기 때문입니다.

다음장에서는 GNU 스몰토크 프로그래밍 언어에 대한 기본 내용과 첫번째 프로그램을 소개하도록 하겠습니다.

연습문제

1. 프로그래밍 언어란 무엇이며, 왜 그것이 필요한가?
2. 컴파일 프로그래밍 언어와 인터프리터 언어의 차이점을 간단히 설명해보라. 각각의 장점과 단점은 무엇인가? 이 경우 GNU 스몰토크는 어떻게 분류할 수 있는가?
3. 프로그래밍 패러다임이란 무엇인가?
4. 프로그래밍 언어가 하나 이상의 프로그래밍 페러다임을 지원할 수 있는가? 다른 언어와 구분되는 GNU 스몰토크의 비밀스런 요소는 하나 이상의 패러다임을 지원하는가?
5. 십진수 543 을 바이너리, 8 진수, 16 진수 형식으로 써라.
6. 바이너리 10110100 을 10 진수, 8 진수, 16 진수 형식으로 써라.
7. 16 진수 A93F 를 바이너리, 8 진수, 10 진수 형식으로 써라.
8. 바이너리 파일과 텍스트 파일은 무엇인가? 이 장에서 제시한 바이너리 파일 외의 다른 종류의 바이너리 파일을 제시할 수 있는가? 파일이 바이너리인지 텍스트 파일인지 구분할 수 있는 방법을 제시할 수 있는가?
9. 텍스트 에디터와 워드프로세서의 차이를 설명하라. 왜 프로그램을 작성하는 데에 워드프로세서를 쓰지 않은 것인가?

2 http://www.flos-freeware.ch/notepad2.html

3 http://www.editpadpro.com/editpadlite.html

불운하게도 현 세대의 메일 프로그램들은 보낸 사람 스스로가 자신이 하고 있는 말을 알고 말하는 것인지 확인하는 기능을 갖추지 못했다.

앤드류 S. 타넨바움 (Andrew S. Tanenbaum)

제 2 장 GNU 스몰토크 소개

이 장에서는 스몰토크의 세계로 들어가서 우리의 첫 번째 프로그램을 작성해보겠습니다. 이 장은 다른 장보다 훨씬 짧습니다. 두 가지 이유가 있는데, 첫 번째 이유는 여러분이 부록 B 에서 필요한 부분을 읽어야하기 때문이고, 두 번째 이유는 여러분의 첫 프로그램을 작성하는 일이 인류에게는 하나의 작은 발걸음에 불과하지만 여러분에게는 엄청난 도약이 될 수 있기 때문입니다.

스몰토크에 대한 짧은 대화(Small Talk)

우리는 프로그래밍을 배우기 위해 스몰토크 프로그래밍 언어를 사용할 것입니다. 그 방법에 있어서 차이가 있을 수 있지만 어떤 프로그래밍 언어로도 결과적으로 컴퓨터를 프로그래밍하는 방법을 가르칠 수 있습니다.

스몰토크는 컴퓨터 과학사에서 매우 특별한 위치를 차지하고 있습니다. 80 년대 초, 제록스 PARC (Xerox PARC, Palo Alto Research Center) 의 제품으로 대중에 공개된 스몰토크는 객체지향 프로그래밍에 대한 이상과 그런 사고를 돕는 문법과 환경으로 인해 기존의 프로그래밍 언어와는 상당히 달랐습니다. 또한 그 당시에는 생소했던 가상 머신 개념을 사용하기도 하였습니다.

> 주의:
>
> 스몰토크는 단순한 프로그래밍 언어가 아닌, 전체 프로그래밍 환경이라고 할 만한 요소들을 가지고 있습니다. 하지만 이런 설명의 반복을 피하기 위해, "스몰토크 프로그래밍 언어"와 "스몰토크 프로그래밍 환경"이란 용어를 동일한 의미로 사용하겠습니다.

흥미롭게도 스몰토크는 오늘날과 같은 그래픽 사용자 인터페이스 (GUI) 를 갖춘 첫번째 프로그래밍 환경이었습니다. 몇몇 창들이 모니터 (현재 윈도우즈 사용자들이 바탕화면이라 통칭하는 데스크탑, 혹은 작업공간) 에 떠 있었고, 스크롤바나 버튼, 메뉴들이 창 안에 있었습니다. 스몰토크 환경은 제록스 PARC 안에서 당시에는 흔하지 않던 네트워킹에도 사용하였습니다.

따라서 스몰토크의 역사를 찾아보면, 시대를 선도하고, 나중에 나타난 다른 프로그래밍 언어에 영향을 미쳤다는 것을 알 수 있습니다. 스몰토크의 몇몇 개념들은 오늘날 컴퓨터 프로그래밍의 필수요소이기도 합니다.

만약 여러분이 이 절에서 본 적 없는 단어들을 보았다면, 걱정할 필요는 없습니다. 앞으로 우리가 알아야 할 개념들을 모두 공부할 예정입니다. 그리고 이렇게 섬세하게 설계하고, 우아하고, 단순하면서도 강력한 프로그래밍 언어를 학습도구로 사용한다는 것이 다른 언어로 공부를 하는 것보다 훨씬 더 쉽게 프로그래밍 개념을 이해하는 데에 도움이 될 것입니다.

스몰토크 뒤의 일반적인 논리

스몰토크는 순수한 객체 지향 프로그래밍 언어이기 때문에, 모든 것을 객체로서 다룹니다. 객체 간의 대화도 객체들에게 *메시지* *message* 를 전달하는 방법을 통해 해결합니다. 예를 들어, 컴퓨터 객체를 만들고, 여러분의 오피스 프로그램을 실행하도록 메시지를 전

달하면, 컴퓨터가 오피스 프로그램을 실행합니다. 아마 컴퓨터 객체를 소유하는 인간이라는 객체를 만들어 두었을지도 모르겠습니다. 그러면 여러분은 그녀로 하여금 그녀의 컴퓨터가 그녀의 오피스 프로그램을 실행시키는 메시지를 보내라고 그녀에게 메시지를 보낼 수 있습니다. 여러분이 만드는 상세함의 수준은 응용프로그램의 필요에 따라 달라집니다. 때때로 여러분은 인간 객체를 만들 필요가 있을 때도 있고, 없을 때도 있습니다. 객체에 대해서는 나중에 더 다루도록 하겠습니다.

> 프로그래밍 유경험자들에게:
>
> C++이나 Objective-C 와 같은 다른 프로그래밍 언어들과 대조적으로 스몰토크는 절차 프로그래밍을 허용하지 않습니다.

첫 프로그램

이제는 첫 번째 프로그램을 두 가지 방법으로 작성해보도록 합시다. 먼저 작성하기 전에, 프로그래밍 환경이 설치되었는지 확인해야 할 것입니다. 만약 설치하지 않았다면, 부록 A 의 내용을 읽고 지금 당장 설치하십시오.

첫 프로그램은 컴퓨터가 터미널 (윈도우즈에서의 용어로는 커맨드 프롬프트, cmd) 에 문자를 출력하도록 만들 것입니다. 이제 여러분의 터미널을 열어서 다음과 같이 입력하십시오.

```
gst
```

```
GNU Smalltalk ready

st>
```

GNU 스몰토크 인터프리터를 실행하면, 인터프리터가 실행할 명령을 기다리고 있습니다. 이 상태를 *인터렉티브 모드* *interactive mode* (상호작용 모드) 라고 합니다. 이제 프로그래밍 할 준비가 되었습니다. 아래의 코드를 입력하고 <Enter> 키를 입력하십시오.

(역주 : 아래 코드 중에 printNl 의 Nl 은 대문자 N 과 소문자 L 입니다. **N**ew**l**ine 을 의미합니다.)

```
'Hello World!' printNl
```

```
'Hello World!'
'Hello World!'
```

> 주의:
>
> 오래된 버전의 GNU 스몰토크를 사용한다면, 다음과 같이 코드의 끝에 느낌표 부호를 붙여줘야 합니다.
>
> `'Hello World!' printNl!.`
>
> 이 책에 나오는 모든 예제를 실행할 수 있는 새로운 버전의 GNU 스몰토크를 설치하시기 바랍니다.

출력에서도 보셨다시피, 컴퓨터는 문자열 'Hello World!'를 (프로그래밍 용어에서는 *표준 출력* *standard output* 이라 불리는) 터미널에 두 번 출력하였습니다. 이제 이 코드에 대해 설명하고, 왜 컴퓨터가 두 번이나 출력했는지 알아보도록 하겠습니다.

설명했다시피, 스몰토크에서 모든 것은 객체입니다. 그리고 객체에 메시지를 전달하고, 각각의 객체에서 응답을 받는 것으로 대화가 이루어집니다. 위의 코드에서는 문자열 객체와 메시지가 있었습니다. 문자열 객체는 작은 따옴표로 문자들을 감싸서 만듭니다. 따라서 `'Hello World!'`는 문자열 객체입니다. 우리는 이 객체에게 스스로 터미널에 출력되기를 원합니다. 따라서 우리는 메시지를 전달해야 합니다. `printNl`은 이러한 목적의 메시지입니다. `"print"` 부분은 아실테고, `"Nl"`은 출력 후에 개행문자를 붙이라는 내용입니다.

내용을 이해하셨어도, 왜 한 번이 아닌 두 번 출력하였는가를 알아야 할 것입니다. 첫번째 출력한 내용은 우리가 프로그래밍 했기 때문에 그러한 것이고, 두번째 출력한 내용은 인터렉티브 모드에서 `'Hello World!'` 값에 대해 전체적으로 수식 표현을 확인하기 위한 것입니다. 보통 터미널 상에 마지막으로 수식 적용이 끝난 결과값을 출력합니다. 만약 바로 값을 썼다면, 그대로 출력될 것입니다. 왜냐하면 값 자체도 수식이기 때문에 항상 확인하는 차원에서 출력을 합니다. 명령어로 수를 입력하는 것도 시도해 볼 수 있습니다. 실제로 수 또한 객체이며, 명백히 그 자체의 값도 확인할 수 있습니다. 명령행에 3 을 입력해보도록 합시다.

```
3
```

```
3
```

보셨습니까? 여러분이 제 말을 믿어주실 수 있길 바랍니다. ^^

즉, 우린 이미 두 개의 컴퓨터 프로그램을 작성하였던 것입니다. 그러나 예전에 말하였듯이 프로그램을 작성하는 방법은 두 가지가 있다고 하였습니다. 그래서 이번엔 다른 방법을 시도하도록 하겠습니다. 이제 텍스트 파일에 프로그램을 기록하고, GNU 스몰토크 인터프리터가 텍스트 파일을 읽어서 프로그램을 실행하도록 해볼 것입니다. 그래서, 이번에는 인터렉티브 모드가 아닌 셈입니다. 여기서 텍스트 파일들을 *소스코드* source code 라고 부릅니다. 이제 여러분이 좋아하는 텍스트 에디터를 열고, 다음 코드를 새 파일로 저장하시기 바랍니다.

```
"hello_world.st"
"A program to print 'Hello World!' to the terminal."

'Hello World!' printNl
```

이제 `hello_world.st` 로 파일을 저장하십시오. 이것은 프로그램의 소스코드입니다. 큰 따옴표 안의 내용은 *주석* comment 입니다. 주석은 두 가지 목적이 있습니다. 하나는 소스코드를 읽는 사람들이 빠르게 이해할 수 있도록, 소스코드의 내용을 깔끔하게 만드는 것입니다. 따라서, 인터프리터는 주석을 생략합니다. 또 다른 목적은 문서화입니다. 어떨 때 주석이 되고 어떨 때 문서화가 되는지는 나중에 설명하겠습니다. 지금은 그냥 주석문으로 이해하시면 됩니다. 소스코드의 첫번째 줄은 권장파일명입니다. 따라서 여

러분이 소스코드 파일이름을 뭘로 지을까 걱정하지 않으셔도 됩니다. (배려심 깊죠?) 두 번째 행의 내용은 이후 프로그램이 어떤 작업을 수행할지 설명한 부분입니다. 이 주석으로 프로그램이 어떤 일을 하는지 상기할 수 있습니다. 주석문은 부가적인 부분이라는 것을 알아 주십시오. 원하신다면 안 쓰셔도 상관 없습니다. 하지만 소스코드에 주석을 남기는 것이 좋은 습관이라는 것을 잊지 말아 주십시오. 왜냐하면 프로그램 작성 후에, 프로그램 제작자로서 몇 달 동안이나 유지보수할 경우에, 프로그램을 왜 그렇게 짰었는지 기억해내야 하는 어려움이 있을 수도 있습니다. 또한 큰 프로젝트에서 다른 사람들과 같이 코딩을 할 때가 있을 때에, 각자에게 소스코드의 내용을 쉽게 이해하기 위해서라도 주석이 필요합니다.

자, 인터프리터가 프로그램을 어떻게 수행할까요? 만약 GNU 스몰토크(*GST*) 프로그램이 아직 실행중이라면, 다른 말로 터미널에서 다음과 같이 표시되고 있다면

```
st>
```

리눅스 사용자는 <CTRL> + d 를, 윈도우즈 사용자는 <CTRL> + z 를 누른 다음 <Enter>키를 눌러서 프로그램을 종료합니다. 다시 GST 프로그램을 시작할 예정이지만, 이번엔 소스코드를 명령행의 인자로 주어 실행할 것입니다. `hello_world.st` 파일이 있는 디렉터리로 이동하여서, 다음 명령어를 입력하시기 바랍니다.

```
gst hello_world.st
```

```
'Hello World!'
```

소스코드에 작성한 코드를 수행한 후, GST 프로그램은 여러분을 텅 빈 터미널에 남겨두고 알아서 자동으로 종료할 것입니다. 알아두실 것은 이번엔 인터렉티브 모드에서 나왔기 때문에, 문자열이 한 번만 출력된다는 것입니다.

소스코드를 파일에 저장하는 데에는 많은 잇점이 있는데, 여러분이 원하는 만큼 실행할 수 있고, 놔뒀다가 다시 실행 할 수 있기 때문에, 프로그램을 모두 다시 타이핑 하지 않아도 된다는 것입니다. 그러나 짧은 작업이나 실험적인 코딩을 하기에는 명령행에 프로그램을 입력하는 것이 간단하기 때문에 배우는 동안 자주 사용할 것입니다.

다음 장에서는 객체, 메시지, 클래스의 개념을 설명할 것입니다.

연습문제

1. 스크린에 다음과 같이 다이아몬드를 출력하는 프로그램을 작성하십시오.

```
  /\
 /  \
/    \
\    /
 \  /
  \/
```

(힌트 : 슬래시와 역슬래시, 공백문자를 이용하십시오.)

2. 프로그램을 소스코드 파일에 쓰는 것은 어떤 장단점이 있습니까? 인터렉티브 모드에서 프로그램을 입력하는 것은 어떤 장단점이 있습니까?

3. GNU 스몰토크 소스 코드 파일들 안에서 주석은 무엇입니까? 주석은 어떻게 입력할 수 있습니까?

4. 컴퓨터의 강력한 점은 대량의 자료를 기억하고, 놀랄 정도로 빠르게, 실수 없이 처리하는 데에 있습니다. 1 부터 1000 까지의 숫자를 스크린에 출력하는 프로그램을 작성할 수 있습니까? 어떻게 하면 쉽게 가능할지 상상해 보십시오.

5. 터미널을 표준 출력이라고도 합니다. 그렇다면 다른 출력 장치들도 있을 수 있습니다. 몇 가지, 예를 들 수 있겠습니까?

적어도 시간의 9 할을 실패하지 않았다면,
당신은 충분히 높은 목표를 설정하지 않은 것이다.

앨런 케이 (Alan Kay)

제 3 장 객체, 메시지, 클래스 : 1 부

객체와 메시지

객체 object 는 프로그램에서 하나의 대상을 나타내는 단위입니다. 객체는 *인스턴스 변수 instance variable* 라고 하는 객체 상태를 가지고 있으며, 외부 (사용자나 다른 객체) 로부터 오는 메시지에 어떻게 응답할지 정의한 선언들, 즉 *메소드 method* 를 가지고 있습니다. 메시지를 작성하고, 객체와 인스턴스, 그리고 메소드를 정의하는 데에는 일관된 문법 규칙이 쓰입니다. 몇 가지 정의를 작성했지만 아마도 여러분 대부분은 제대로 이해하지 못했을 것입니다. 만약 그러하다면, 윗 문단을 한 번 더 읽고 진행하기 바랍니다. 왜냐하면 처음 접하는 경우라면 이러한 개념을 갖기가 어렵기 때문입니다. 앞으로 나올 예제들은 이러한 개념을 확실하게 해줄 것입니다.

이제 몇몇 예제 객체와 메시지 표현들을 보도록 하겠습니다. 먼저 수식표현은 어떨까요? 아래의 코드를 GNU 스몰토크 인터프리터에 입력하고 키보드에서 <Enter>키를 눌러보십시오.

```
3 + 4
```

```
7
```

공백이 있는지 여부는 그리 중요하지 않습니다. 원하는 만큼 긴 공백을 넣을 수도 있고, 공백을 안 넣을 수도 있습니다. 덧셈 부호 앞뒤로 개행문자를 넣을 수도 있습니다. 이제부터 공백이라 불리는 개념을 설명하겠습니다.

공백 white space 은 빈 칸, 탭, 개행문자에 주어진 이름입니다. 모두 다 공백이라 부르는데, 당연히 보이기 않기 때문에 그렇게 부릅니다. 보통 공백은 프로그래밍 언어에서 의미가 있는 부분이 아니기 때문에, 컴파일러와 인터프리터들은 문자열 안에 들어있는 경우가 아니라면 공백을 무시합니다.

이제 수학적 표현으로 돌아와봅시다. 여기서 주 객체는 3 입니다. 우리가 메시지를 보낼 객체는 3 이며, 메시지를 받기 때문에 3 을 *리시버 receiver* 라고 부릅니다. + 4 라는 문자들은 메시지가 되어 리시버 3 에게 날아갑니다. 3 에 보낸 메시지는 다음과 같습니다. "내가 보낸 숫자와 너 자신을 더한 후에 결과를 반환하라."

선택자 (selector) 와 인자 (argument)

이제는 위에서 보낸 메시지를 자세히 보도록 하겠습니다. 실제 이 메시지에서는 두 부분이 있습니다. 하나는 선택자이며 + 문자입니다. 또 다른 부분은 인자로 객체인 4 입니다. 여러분은 메시지의 인자 부분을 제거하는 것으로 선택자를 얻을 수 있습니다 (이에 대한 예제는 나중에 보여드리겠습니다).

선택자는 메시지에 어떻게 응답할지 결정하도록, 객체를 돕는 체계를 형성합니다. 예를 들어, 여기 + 선택자는 실제로 객체 3 안에 정의되어 있으며, 3 은 이 메시지를 받았을 때 어떻게 할지 알고 있습니다.

인자 argument 는 객체가 응답을 실행하는 동안 사용해야 하는 추가적인 정보입니다. 객체

는 이들 정보를 응답을 실행할 때 어떻게 사용할지를 알고 있습니다. 객체가 이 모든 것을 어떻게 아는지는 우리가 원하는 객체를 어떻게 정의할 수 있는지를 알아보면서 보게 될 것입니다. 주의할 점은 메시지가 꼭 인자를 가질 필요는 없다는 점입니다. 실제로 다음의 메시지는 아무런 인자를 보내지 않습니다. 여기를 보십시오.

```
'Hello World!' size
```

```
12
```

여기 문자열 객체 `'Hello World!'`와 메시지 `size`가 있습니다. 이 메시지는 문자열 객체에게 갖고 있는 문자열의 길이, 즉 얼마나 많은 문자들을 가지고 있는지를 반환하라는 내용입니다. 보시다시피 `size` 메시지는 따로 인자가 필요하지 않기 때문에, 어떤 인자도 가지고 있지 않습니다. 즉 메시지가 모든 정보를 가지고 있으며, 다른 말로는 질문 속에 답이 있는 것입니다. 좋습니다. 이 정도면 설명이 충분하겠죠?'

바로 이전의 예제를 보고 메시지 형식에 대해 말해보겠습니다. 컴퓨터는 오로지 1 과 0 을 이해하고 있다고 말 했습니다. 따라서 숫자 3 은 실제로 컴퓨터 메모리 안에서 다음과 같이 2 진수로 기록되어 있습니다[4].

```
00000011
```

우리는 8 자리로 표시하는 1 바이트의 메모리에 이와 같이 기록하였습니다. 이제 위에서 주어진 정보를 다음 코드를 입력해봅시다.

```
3 bitAt: 3 put: 1
```

```
7
```

이 표현은 객체 숫자 `3`과 `bitAt: 3 put: 1`이라는 메시지를 가지고 있습니다. 이 메시지에는 `bitAt: put:`이라는 선택자와 3 과 1 이라는 인자를 가지고 있습니다. 어떻게 선택자의 이름을 얻는지 기억하고 계십니까? 인자를 제거한 메시지의 나머지 부분이 선택자라고 하였습니다.

이 메시지는 정수 객체 3 에게 세번째 비트 값을 1 로 대체하라고 말하고 있습니다. 비트 자릿수는 오른쪽부터 왼쪽으로 세어 나갑니다. 3 의 세번째 비트는 0 입니다. 그러나 이제는 1 입니다. 따라서 우리는

```
00000111
```

위와 같이 십진수로 7 에 해당하는 값을 얻을 수 있습니다. 메시지를 보낸 후, 수식 표현을 평가해서 터미널에 출력합니다.

단일항 메시지, 이항 메시지 그리고 키워드 메시지

메시지 형식은 3 가지가 있습니다. 각 형식 간 차이점은 선택자, 인자, 그리고 우선순위

4 실은 정수를 메모리에 저장하는 형식은 성능 상의 문제로 더 복잡합니다. 그러나 여기서는 예를 위해 이렇게 가정한 것입니다.

와 관련한 것입니다. 우선순위가 무엇이냐고요? 그건 조금 있다가 설명드리겠습니다.

단일항 메시지 *unary message* 는 인자가 없는 메시지입니다. 단일항이라 불리는 것은 결과적으로 표현식이 객체 하나만을 포함하고 있기 때문입니다. 단일항 메시지의 예를 보여드리겠습니다.

```
'Hello World!' size
```

보시다시피 여기엔 인자가 없으며, 전체 표현은 문자열 객체 `'Hello World!'` 만을 포함하고 있습니다.

이항 메시지 *binary message* 는 하나의 인자를 가진 메시지입니다. 이항 메시지의 다른 특성은, 선택자에는 알파벳이나 숫자가 아닌 문자로 2 자까지 가능합니다. 예를 들어, 다음의 이항 메시지를 보십시오.

```
3 + 4
```

여기서 + 는 선택자이며, 4 는 선택자에 대한 인자입니다. 표현은 3 과 4, 두 개의 객체를 포함하고 있습니다. + 는 한 개의 문자이며 알파벳이나 숫자가 아닙니다. 따라서 이항 메시지로서 요구사항을 맞출 수 있습니다.

마지막으로 *키워드 메시지* *keyword message* 는 선택자의 인자를 하나 이상 갖는 메시지입니다. 인자는 알파벳, 숫자로 이루어져 있습니다. 선택자의 뒤에 콜론을 붙인 후, 인자가 필요하다는 것을 표시하여야 합니다. 이항 메시지 표현의 예는 다음과 같습니다.

```
3 bitAt: 3 put: 1
```

여기서 선택자는 `bitAt:put:`이며 3 과 1 이 인자입니다. 전체 표현은 세 개의 객체를 포함하고 있지만, 한 개 혹은 그 이상의 객체를 가질 수 있습니다.

메시지 형식에서 선택자와 인자에 대해 얘기하였지만, 아직 우선순위에 대해서는 설명하지 않았습니다. 우선순위는 하나의 표현에 많은 메시지가 있을 때, 어떤 것을 먼저 실행할지 결정하는 개념입니다. 그러한 표현을 상상하기 어렵다면, 걱정하지 마십시오. 여기에 예제가 있습니다.

```
3 + 5 bitAt: 3 put: 1 printNl
```

보시다시피, 여기엔 세 개의 메시지가 있습니다. `+, bitAt:put:` 그리고 `printNl` 까지. 여기서 어떤 것을 제일 먼저 수행해야 할까요? 우선순위 규칙에 따라 해보지요.

스몰토크에는 메시지 우선순위에 대한 몇 가지 기본 규칙이 있습니다.

1. 단일항 메시지는 가장 먼저 실행됩니다.
2. 이항 메시지는 두번째로 실행됩니다.
3. 키워드 메시지는 마지막에 실행됩니다.

4. 만약 괄호 안에 표현이 있으면, 제일 먼저, 그것부터 실행됩니다.

5. 만약 같은 선행 조건을 가지고 있는 메시지가 있다면, 왼쪽부터 오른쪽순으로 메시지가 각각 실행됩니다.

이제 이 우선순위 조건을 설명할 두 가지 예제를 보여드리겠습니다. 첫 번째 예는 위에 제시한 예입니다.

```
3 + 5 bitAt: 3 put: 1 printNl
```

```
1
12
```

이제 왜 1 과 12 를 출력하였는지 볼 시간입니다. 이 표현식에서 인터프리터가 먼저 만나는 것은 단항 메시지 `printNl` 입니다. 그리고 바로 앞의 객체로 이 명령을 수행합니다. 이 경우 1 입니다. 따라서 처음 출력은 1 입니다. 그리고 이 표현식은 1 을 확인하여 그 자체를 답합니다. 이제 표현식은 다음과 같이 정리할 수 있습니다.

```
3 + 5 bitAt: 3 put: 1
```

이제 두 개의 메시지가 있습니다. 하나는 +로 이항 메시지이며, 다른 하나는 `bitAt:put:`으로 키워드 메시지입니다. 규칙에 따라 위 표현식은 8 이라는 결과를 내놓는 이항 메시지를 먼저 실행합니다. 따라서 표현식은 다음과 같이 나타낼 수 있습니다.

```
8 bitAt: 3 put: 1
```

이제 키워드 메시지 하나만 남았습니다. 이 메시지는 정수 12 에 대해 확인할 것입니다 (여러분이 직접 종이에 적어서 계산해보시길 바랍니다). 마지막으로 인터프리터가 터미널 상에 객체를 출력할 것입니다.

이제 두 번째 예제를 보도록 하겠습니다.

```
(3 + 5 bitAt: 3 put: 1) printNl
```

```
12
12
```

이전 예제와 한 가지 다른 점은 `printNl` 메시지 앞의 내용을 괄호로 묶어준 것입니다. 그러나 결과는 아주 다릅니다. 왜냐하면 괄호 안의 메시지를 먼저 실행한 후, `printNl` 메시지를 전달하기 때문입니다.

터미널에 출력을 표시하는 다른 방법

다음 절로 진행하기 전에, 터미널에 텍스트를 출력하는 다른 방법을 보여줄 것입니다. 이 방법은 printNl 메시지와 함께 책 전체에 걸쳐 자주 사용할 예정입니다.

문자열 객체에게 printNl 이라는 메시지를 보내서 자신을 터미널에 출력하라고 명령했지만, 이 과정을 역으로 생각해서, 터미널에게 문자열 객체를 주면서 이것의 값을 출력하라고 메시지를 보낼 수 있습니다. 비록 다른 개념에서 표준출력이라는 이름이지만, 표준출력은 스몰토크에서 Transcript 라는 이름을 갖고 있습니다. 그리고 텍스트를 표시하기 위한 Transcript 가 가지고 있는 메시지 선택자는 show: 입니다. 선택자의 끝에 콜론이 붙는 것으로 생각해볼 때, show: 선택자는 표시하기 원하는 문자열 객체 인자를 필요로 합니다. 여기 gst 명령어를 통하여 GNU 스몰토크 인터프리터가 수행할 예제가 있습니다. 아래의 코드를 입력한 후, <Enter>를 치십시오.

```
Transcript show: 'Hello World!'
```

```
Hello World!
```

문자열 객체에서 printNl 메시지를 사용했던 것과는 약간 다른 점을 알아차릴 수 있을 것입니다. 예를 들어, 이번에는 출력 주변에 작은 따옴표가 없습니다. 또한 출력 다음에 줄바꿈도 일어나질 않았습니다. 조금 있다가 메시지 캐스캐이딩 부분에서 Transcript 를 사용하였을 때, 출력 끝에 개행 문자를 출력하는 어떻게 추가할 수 있을지 보여드리겠습니다. 마지막으로, 이 printNl 메시지를 사용했을 때 두 번 출력되던 것이, 한 번만 출력되었다는 것입니다. 그 이유는 반환되는 객체가 Transcript 자신이고, Transcript 는 문자열의 표현이 없기 때문입니다.

메시지 연쇄

객체 이름에 하나 이상의 메시지를 쓴다면, 모든 메시지들은 이전 메시지의 결과 객체에 전달될 것입니다. 이것을 *메시지 연쇄* *message chaining* 라 합니다. 메시지 연쇄에 대한 예는 다음과 같습니다.

```
objectName message1 message2 message3 ... messageN
```

따라서 `message1` 이 바로 `objectName` 에 전달될 때, `message2` 는 `objectName message1` 표현식의 결과값으로 나오는 객체에 전달됩니다. 같은 논리로 `message3` 는 표현식 `objectName message1 message2` 의 결과 객체에 전달된다고 말할 수 있습니다.

여기 메시지 연쇄가 실제 쓰이는 예제를 보겠습니다.

```
'Canol' reverse asUppercase
```

```
'LONAC'
```

여기서 `Canol` 문자열은 결과 문자열 객체인 `'LONAC'` 에 도달할 때까지 2 번의 메시지를 통해 생성하였습니다.

메시지 캐스캐이딩

GNU 스몰토크는 객체의 이름을 매번 쓰지 않고도 여러 메시지를 전달할 수 있도록 허용하고 있습니다. 이것을 메시지 *캐스캐이딩* *message cascading* 이라고 합니다. 메시지 캐스캐이딩에 대한 문법은 다음과 같습니다.

```
objectName message1; message2; message3; ... ; messageN
```

우리가 보내고자 하는 메시지 사이에 세미콜론을 넣기만 하면 됩니다. 세미콜론이 없는 메시지 연쇄와는 다릅니다. 메시지 연쇄의 경우 모든 메시지는 이전 메시지의 결과 객체로 보내집니다.

여기 메시지 캐스캐이딩의 예제가 있습니다.

```
Transcript show: 'Canol'; cr
```

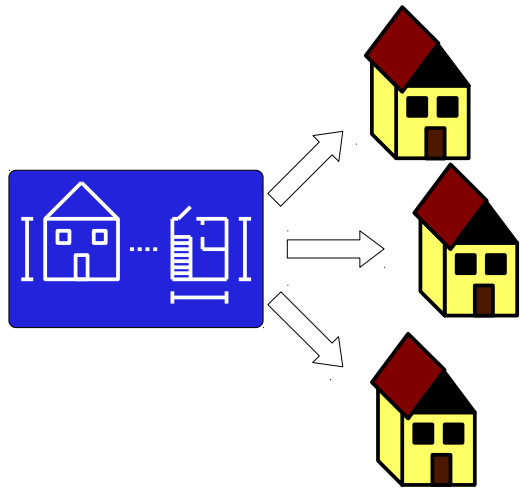
`cr` 은 carriage return 이라고 불리는 메시지로 `Transcript` 객체가 개행을 입력하도록 하는 메시지입니다. 따라서 `Transcript` 객체에 첫 번째 메시지 `show: 'Canol'` 을 보내고, 다음에 `cr` 을 보냅니다.

메시지 캐스캐이딩을 사용할 때에는 주의해야 합니다. 왜냐하면 가끔 우선순위 규칙에 따라 여러분이 예상치 못한 결과가 나올 수 있습니다.

클래스

클래스 *Classes* 는 만들기 원하는 객체의 틀(templates)로 요약할 수 있습니다. 클래스를 정의한 다음, 클래스로부터 객체를 생산합니다. 이것이 객체가 클래스의 인스턴스 Instance 라고 불리는 이유입니다. 이것은 마치 집의 청사진과 같은 것입니다.

모든 객체는 관련된 클래스를 가지고 있습니다. 예를 들어, 'Hello World!' 객체는 String 클래스로부터 생성하였고, 2 나 3 은 SmallInteger 클래스로부터 생성하였습니다. 5 장 이후에 클래스의 이론에 관하여 더 자세히 다루겠습니다.



청사진을 집의 클래스로, 실제 집을 클래스의 인스턴스로 생각할 수 있습니다.

일반 클래스와 사용법 : 1 부

GNU 스몰토크는 풍부한 클래스의 집합을 제공하는 라이브러리를 가지고 있습니다. 이 장에선 가장 흔하게 쓰이는 몇몇 클래스들과 클래스의 인스턴스로 보낼 수 있는 메시지들을 보도록 하겠습니다. 수 백개의 클래스와 수 천개의 메시지가 있지만, 그것을 모두 다룰 수 없습니다. 그래도 많은 에제를 들어 GNU 스몰토크 코드에 친숙할 수 있도록 노력하겠습니다.

수

수는 컴퓨팅에서 가장 공통적으로 사용하는 자료형식 중 하나입니다. 컴퓨터들은 인간보다 빠르고 정확하게 큰 수를 만들고 다룰 수 있습니다. GNU 스몰토크는 이러한 작업을 하는 데에 많은 이점을 줄 수 있는 Number 클래스를 갖고 있습니다.

GNU 스몰토크에서 어떻게 수를 표현할까요? 여기에 수의 몇 가지 표현이 있습니다.

자연수를 종이에 적듯이 표현할 수 있습니다.

```
3

15

1000000
```

음수로 만들 때에는 자연수 앞에 하이픈 문자를 추가합니다.

```
-3

-15

-1000000
```

소수를 표현할 때에는 점을 추가합니다.

```
3.5

-15.7

0.1
```

다른 진법의 수도 표현할 수 있습니다. 단순히 기수를 (십진수로) 추가하고, 수의 앞에 r 을 붙이면됩니다. (십진수 이상의 진수에서 9 보다 더 큰 수를 표현하기 위해 A 부터 시작하는 문자를 사용합니다.)

`8r312` (십진수 202)

`16r312` (십진수 786)

`16rABC` (십진수 2748)

GNU 스몰토크에서는 지수표기법도 사용합니다. 지수표기법에서 e 문자는 *십의 지수*로 사용합니다. 따라서 십의 지수를 e 문자 뒤에 써서 표현합니다.

`1e2` (100 과 같음)

`0.3e5` (30000 과 같음)

`0.0015e4` (15 과 같음)

GNU 스몰토크에서는 두 개의 정수를 슬래시 문자 좌우에 두는 것으로 분수를 표현할 수 있습니다. 분수는 자동으로 가능한한 단순표기됩니다.

```
3/5
```

```
5/3
```

이제 우리가 사용할 수를 어떻게 표현하는지 알았으니, 수식을 이용하여 수를 써봅시다. 기본 수학 연산으로 시작하는 것이 좋겠습니다.

+

덧셈을 위한 이항 메시지 + 가 있습니다.

```
3 + 5
```
```
8
```

```
0.3 + 0.01
```
```
0.31
```

-

하이픈은 뺄셈에 씁니다.

```
5 - 3
```
```
2
```

```
3 - 5
```
```
-2
```

*****

* 문자(x 문자가 아님)는 곱셈에 사용합니다.

```
2 * 3
```
```
6
```

```
0.1e-2 * 1e2
```
```
0.1
```

/

슬래시 문자는 나눗셈에 사용합니다. 정확하게 표현하기 위해 분수나 소수를 전달합니다.

```
50/4
```

```
25/2
```

```
0.5/5
```

```
0.1
```

\\

두 개의 역슬래시 문자는 나머지 연산을 위해 사용합니다.

```
5 \\ 3
```

```
2
```

```
3 \\ 3
```

```
0
```

between:and:

두 개의 수 사이에 해당하는 수가 존재하는지 여부를 `between:and:` 선택자를 이용하여 제어할 수 있습니다. 예를 들어, 3 이 1 과 5 사이에 있는지 알아보려면 다음과 같습니다.

```
3 between: 1 and: 5
```

```
true
```

여러분도 보시다시피 GST 에서 `true` 라는 이름의 객체를 반환하였습니다. 이 객체는 `Boolean` 클래스의 인스턴스로 논리적 명령의 참/거짓을 결정하기 위해 사용합니다. 그렇다면 논리적 거짓은 어떻게 표현할까요?

```
3 between: 4 and: 6
```

```
false
```

여러분도 보시다시피 false 라는 이름의 객체를 표시하였습니다. 4 장, 예외 흐름 제어에서 이 Boolean 객체를 어떻게 사용하는지 더 상세히 설명하도록 하겠습니다.

abs

음수의 절대값을 얻기 위해 abs 메시지를 사용합니다.

```
-3 abs
```

```
3
```

degreesToRadians

`degreesToRadians` 메시지를 사용하여 각도 크기를 라디안 값으로 변환할 수 있습니다. 이 연산은 삼각함수 계산에 유용하게 사용합니다.

```
180 degreesToRadians:
```

```
3.141592653589793
```

GST 는 파이를 무척 빠르고 정확하게 계산하지 않습니까?

cos

`cos` 메시지를 사용하여 각의 코사인 값을 계산할 수 있습니다.

```
180 cos
```

```
-0.5984600690578581
```

잠깐, 180 의 코사인 값은 -1 아니던가요? 단위에 주의하시기 바랍니다. `cos` 메시지는 각도의 크기보다는 라디안의 크기를 보내주길 원합니다. 따라서 먼저 `degreesToRadians` 로 각도를 라디안 값으로 변환한 다음에 사용해야 합니다. 조금 있다가 `degreesToRadians` 메시지의 결과를 어떻게 cos 메시지로 보내는지 설명하겠습니다.

negated

`negated` 메시지는 수의 음수값을 찾는데 사용합니다.

```
3 negated
```

```
-3
```

주의하십시오. 만약 이미 음수인 값에 이 메시지를 보내면, 양수값을 얻게 될 것입니다.

```
-3 negated
```

```
3
```

raisedTo:

`raisedTo:` 메시지는 수의 n 승 값을 구하는 데에 사용합니다.

```
3 rasiedTo: 4
```

```
81
```

아시다시피 어떤 수의 0 승은 항상 1 입니다.

```
3 raisedTo: 0
```

```
1
```

GST 는 멋진 분수로 음의 멱승을 계산하여 줍니다.

```
3 raisedTo: -2
```

```
1/9
```

squared

`raisedTo: 2` 메시지 대신에 `squared` 메시지를 사용하여 제곱승을 구할 수 있습니다.

```
3 squared
```

```
9
```

even

짝수인지 아닌지 `even` 메시지를 이용하여 알아낼 수 있습니다.

```
4 even
```

```
true
```

```
2222222221 even
```

```
false
```

odd

그리고 또한 홀수인지 아닌지 `odd` 메시지를 이용하여 알아낼 수 있습니다.

```
3 odd
```

```
true
```

sign

`sign` 메시지는 양수라면 1, 음수라면 -1 을 출력합니다.

```
6 sign
```

```
1
```

```
-0.3
```

```
-1
```

이제 우리는 부동소수점을 사용하는 몇 개의 메시지를 볼 것입니다.

integerPart

`integerPart` 메시지는 부동소수점의 정수부분을 알려줍니다.

```
0.7 integerPart
```

```
0.0
```

```
3.1 integerPart
```

```
3.0
```

truncated

십진수의 버림수를 얻기 위해 `truncated` 메시지를 사용합니다.

```
17.2 truncated
```

```
17
```

이것은 반올림이 아닙니다.

```
17.6 truncated
```

```
17
```

반올림을 하기 위해서는...

rounded

`rounded` 메시지를 사용합니다.

```
17.6 rounded
```

```
18
```

분수에 대한 특별한 메시지들을 봅시다.

denominator

분모를 얻기 위해서 `denominator` 메시지를 사용합니다.

```
(3/4) denominator
```

```
3
```

주의할 점은 3/4 에 괄호를 붙여서 썼다는 것입니다. 왜냐하면 만약 괄호를 안 썼을 경우, `denominator` 메시지가 단일항 메시지이기 때문에, 3/4 표현을 계산하기 전에 `denominator` 메시지가 처리되었을 것입니다. 이전에도 말하였지만, 단일항 메시지는 이항 메시지보다 먼저 처리하도록 되어있습니다.

numerator

분수의 분자를 얻기 위해 numerator 메시지를 사용합니다.

```
(3/4) numerator
```

```
4
```

setNumerator:setDenominator:

분수의 분자와 분모를 지정하기 위해, `setNumerator:setDenominator:` 메시지를 사용합니다.

```
3/4 setNumerator: 5 setDenominator: 6
```

```
5/6
```

이번에는 괄호를 사용하지 않았는데, 이번에는 키워드 메시지를 3/4 표현에 보내기 때문입니다. 이항 메시지는 키워드 메시지보다 먼저 처리하기 때문입니다.

문자

문자는 자료를 표현하는 단일 기호입니다. 예를 들어, 글자, %와 같은 특수문자, 9 와 같은 숫자 등은 모두 문자입니다. 스몰토크에서 문자는 개별 객체입니다. 스몰토크에게 어떤 기호를 문자라고 인식시키려면, 그것 앞에 $를 붙여야 합니다. 예를 들어서

```
$a

$%

$9
```

와 같이 표현하며, 달러 기호도 문자로 사용하려면 다음과 같이 입력합니다.

```
$$
```

문자를 다루기 위한 방법은 많으며, 그 중 몇 가지를 아래에 설명하겠습니다.

asLowercase

만약 문자들을 소문자로 바꾸길 원한다면, `asLowercase` 메시지를 사용하십시오.

```
$D asLowercase
```

```
$d
```

asUppercase

`asLowercase` 의 반대 작용을 하는 메시지입니다.

```
$d asUppercase
```

```
$D
```

isAlphaNumeric

만약 기호가 알파벳, 숫자인지 확인하려면, `isAlphaNumeric` 메시지를 사용할 수 있습니다.

```
$% isAlphaNumeric
```

```
false
```

```
$3 isAlphaNumeric
```

```
true
```

사용자로부터 입력을 받을 때 이러한 방법을 필요로 할 것입니다. 예를 들어 사용자가 어떤 데이터를 입력할지 알 수 없을 때에는, 이러한 방법을 사용하여서 유효한 형식인지 분별하는 데에 사용할 것입니다.

isDigit

기호가 숫자인지 확인하기 위한 메시지입니다.

```
$4 isDigit
```

```
true
```

```
$c isDigit
```

```
false
```

isLetter

글자(알파벳) 인지 아닌지 여부도 관심있을 것입니다.

```
$c isLetter
```

```
true
```

```
$4 isLetter
```

```
false
```

문자열

문자열은 여러 개의 문자객체들의 조합으로 이루어진 객체입니다. 단어, 문장, 문단 등을 표시할 때 사용합니다. 스몰토크에 문자열을 생성한다고 알려야 할 때에는 '(작은 따옴표)문자를 문장의 앞과 뒤에 붙여야 합니다. 예를 들어,

```
'Canol'

'I want a cup of coffee'
```

는 모두 문자열 객체입니다. 문자열 내에서 작은 따옴표를 표현하고 싶을 때에는 두 개의 작은 따옴표를 연속으로 사용합니다.

```
'Eiffel Tower''s height varies as much as six inches, depending on
the temperature.'
```

큰 따옴표가 아니라, 2 개의 작은 따옴표라는 것에 주의하여 주십시오.

프로그래머들이 문자열을 엄청나게 쓰기 때문에, 문자열에 관한 여러가지 유용한 방법들이 많습니다. 아래 선택한 몇몇 메시지들을 봅시다.

includes:

문자열 안에 지정한 문자가 들어있는지 여부를 확인하기 위해 `includes:` 메시지를 사용합니다.

```
'Canol' includes: $n
```

```
true
```

주의할 점은 대소문자를 가린다는 것입니다.

```
'Canol' includes: $N
```

```
false
```

indexOf:

가끔은 문자가 문자열에 있는지 여부를 아는 것만으론 충분치 않을 때가 있습니다. 어디에 있는지도 알아야 할 때가 있습니다. `indexOf:` 메시지는 어디에 메시지가 있는지 알려줄 수 있습니다.

```
'Canol' indexOf: $n
```

```
3
```

만약 문자가 존재하지 않는다면, 이 메소드는 0 을 반환합니다. 문자가 하나 이상 있을 경우, 문자열에서 처음 나타난 위치를 반환합니다.

reverse

제가 스몰토크에서 좋아하는 메시지 중 하나입니다. `reverse` 메시지를 통해 문자열을 거꾸로 씁니다.

```
'.dlrow eht ni remmargorp tseb eht si lonaC' reverse
```

```
'Canol is the best programmer in the world.'
```

재밌나요?

countSubCollectionOccurrencesOf:

다른 문자열에서 해당 문자열이 몇 번이나 있는지 알아보는 메시지입니다.

```
'Thomas Edison, the inventor of the light bulb, was afraid of the
dark.' countSubCollectionOccurrencesOf: 'the'
```

```
3
```

,

만약 두 개의 문자열을 하나의 String 객체로 만들기 원한다면, ,(콤마) 문자를 문자열 사이에 두십시오.

```
'Best', ' friends'
```

```
'Best friends'
```

실제로 여러분이 연결하기 원하는 문자열들을 원하는 만큼 붙일 수 있습니다.

```
'Best', ' friends', ' should', ' never', ' be', ' separated.'
```

```
'Best friends should never be separated.'
```

asUppercase

asUppercase 메시지를 사용하여 문장 전체를 대문자로 만들 수 있습니다.

```
'Hey I''m talking to you!' asUppercase
```

```
'HEY I''M TALKING TO YOU!'
```

그리고 `asLowercase` 메시지는 문장을 소문자로 변환할 것입니다. 일부러 예제를 드리진 않겠습니다.

size

`size` 메시지는 문자열에서 문자가 몇 개 있는지 알려주는 메시지입니다.

```
'There are 39 characters in this string.' size
```

```
39
```

변수

> 스몰토크에서의 변수는, 수학에서의 변수와 매우 흡사합니다. 이 주제를 읽는 동안 마음 속에 기억해두면, 이 개념을 쉽게 상상할 수 있을 것입니다.

프로그램을 작성하는 동안, 수많은 객체를 씁니다. 객체를 생성하고, 다루고, 파괴합니다. 그러나 객체를 생성한 후, 프로그램의 어떤 부분에서 참조하려 할 때 어떻게 해야 하겠습니까? Human 클래스(스몰토크에서 제공하는 클래스가 아니라, 가상의 클래스라고 생각하시면 됩니다.)를 가지고 있다고 가정하겠습니다. 이 클래스로부터 새로운 객체를 만들 것입니다. 실제 세계의 인간처럼, 각각의 객체는 이름이 있어야, 우리가 그들과 대화할 수 있을 것입니다. 우리가 메시지를 전달하여 대화를 할 것입니다. 메시지를 전달하기 전에 메시지를 받길 원하는 객체를 지정해야 할 것입니다. 객체를 반환하는 표현식이나 객체의 이름으로 그것을 할 수 있습니다.

Human 인스턴스 Carl 을 가지고 있다고 가정하겠습니다. 우리는 그에게 이렇게 메시지를 보낼 수 있습니다.

```
Carl tellUsYourLastName
```

여기서 Carl 은 Human 인스턴스를 참조합니다.

객체에 이름 지어주는 것을 *지정* *assignment* 이라고 합니다. 그리고 지어준 이름을 프로그래밍 용어로 *변수* *variables* 라고 합니다. (변할 변자를 쓰는) 변수라는 용어를 쓰는 이유는, 한 객체만을 참조할 수 있지만, 그것이 언제든지 프로그램의 다른 객체를 가리킬 수 있기 때문입니다. 객체에 이름을 어떻게 지정하는지 배우기 전에, 먼저 왜 그러한 것을 원하게 되는지 얘기해보겠습니다. human 객체를 만들었던 위의 예제같은 경우에는, 아마도 그 변수가 다른 객체를 참조하도록 바꾸길 원치 않을 것입니다. 그러나 다른 예제로, 하루에 식당에 방문한 사람들의 수를 추적하는 프로그램을 가지고 있다고 가정했을 때, 손님이 올 때마다 숫자는 증가할 것입니다. 그렇게 하려면 어떻게 해야 되겠습니까? 먼저 객체 0 을 참조하는 변수를 생성합니다. 그리고 손님이 올 때마다 변수를 호출하여, 그 변수가 가리키는 객체를 가져다가, 1 을 더하여 변수가 그것을 가리키게합니다. 따라서 매회 손님들이 올 때마다 변수는 다른 객체를 참조할 것입니다.

이제 변수 할당 문법을 알아볼 시간입니다. 할당을 하기 전에, 변수를 만들어야 합니

다. 변수를 만드는 기본 구조는 다음과 같습니다.

```
| aVariable anotherVariable andAnotherVariable |
```

여기 `aVariable`, `anotherVariable` 그리고 `andAnotherVariable` 이라고 이름 붙여진 세 개의 변수를 생성하였습니다. 우리는 파이프 문자 사이에 만들기 원하는 변수 이름을 두었습니다. 변수 이름은 빈 칸으로 구분하였고, 우리가 원하는 만큼 변수를 생성할 수 있습니다. 변수 지정은 다음과 같이 할 수 있습니다.

```
aVariableName := anObject
```

지정 연산자 *assignment operator* 라고 부르는 연산자는 콜론과 등호 기호를 빈칸 없이 연결하여 구성합니다. 이 연산자는 우측 변의 객체를 좌측 변의 변수에 할당합니다.

모든 객체는 컴퓨터 메모리의 공간을 차지하도록 생성하며, 변수의 역할은 실제로 객체의 메모리 주소를 가지고 있는 것입니다.

사용자 입력 얻기

이 절에서는 사용자의 표준 입력 장치(아마 대부분 키보드)를 통하여 사용자로부터 정보를 얻어오는 코드를 설명하겠습니다. 만약 여러분이 전에 콘솔 프로그램을 사용해봤다면, '이것을 실행하기 원하십니까? (y/n)" 같이 y 나 n 을 입력하기 원하는 질문을 본 적이 있을 것입니다. 이것은 터미널로부터 사용자의 입력을 얻는 기술을 사용한 것입니다.

```
"user_input.st"
"A program to demonstrate how to get input from user."

| userName |

Transcript show: 'What is your name? '.
userName := stdin nextLine.

Transcript show: ('Hello ', userName, '!'); cr.
```

```
What is your name? Canol
Hello Canol!
```

이 프로그램은 사용자에게 이름을 물어보고, 이름을 부르며 인사를 하는 프로그램입니다. 사용자 입력을 얻는 부분은 `userName := stdin nextLine` 입니다. `stdin nextLine` 은 개행문자로 키보드 입력을 마치는 키보드 입력을 기다린다는 내용입니다. 이 경우, 사용자는 원하는 내용을 입력하고 <Enter> 키를 누르는 것으로 입력 과정을 마칩니다. `stdin nextLine` 은 입력 문자열을 반환하고, 그 값을 `userName` 변수 안에 할당합니다. 할당 연산자는 객체에 보내는 다른 종류의 메시지들에 비해 순서가 낮습니다. 따라서 `stdin nextLine` 이 수행되고, 나중에 할당을 하게됩니다.

공통 클래스와 사용법 : 2 부

이제 여러분들이 아마 사용하게 될 다른 유용한 클래스들을 소개하겠습니다. 예제에서

는 변수 개념도 사용할 것입니다.

배열

여러 종류의 자료와 작업들을 처리할 때, 그룹화 할 필요가 있습니다. 보통, 우리가 그룹화한 데이터들은 특정 방식으로 연결되어 있습니다. 장보기 목록이나 우리가 좋아하는 영화, 읽었던 책 등 처럼 말입니다.. 따라서 다른 객체의 집합을 갖고 있을 수 있는 객체가 필요합니다. Array, Set, Dictionary 클래스는 다른 방법으로 여러 객체를 한꺼번에 다루는 객체들입니다.

먼저 Array 클래스를 보겠습니다. 이 클래스는 지정한 수 만큼의 객체들을 순서대로 갖고 있을 수 있습니다.

new:

`new:` 메시지를 Array 클래스에 전달하여 새 Array 객체를 만들 수 있습니다. 얼마나 많은 객체를 배열에 둘 것인지 정해야만 합니다. 코드는 다음과 같습니다.

```
| anArray |

anArray := Array new: 10
```

```
(nil nil nil nil nil nil nil nil nil nil )
```

위의 내용을 각각의 행으로 입력하거나, 파일에 소스코드를 입력한 후 GST 인터프리터에 전달하는 것으로 프로그램을 실행할 수 있습니다만 앞으로 작성할 내용을 위해, 각각의 행으로 입력하여주십시오.

각각의 행을 입력하였다면, 이제 `anArray` 변수가 다른 열 개의 객체를 담을 수 있는 Array 객체를 참조할 것입니다. 현재의 Array 객체의 상태를 위에서 볼 수 있을 것입니다. 먼저 Array 객체의 모든 객체는 아무것도 나타내지 않는 특수한 객체 nil 로 초기화하였습니다.

at:

어떤 한 위치의 객체를 알아보기 위해 `at:` 메시지를 사용할 수 있습니다. 입력한 프로그램 내용을 유지한 상태에서, 아래와 같이 명령을 입력하여 주십시오.

```
anArray at: 1
```

```
nil
```

물론 현재 배열의 첫 번째 객체는 nil 입니다. 여기서 배열의 위치로 전달한 값 1 은 인덱스(indexes)라고 부릅니다. 스몰토크의 인덱스는 1 로 시작합니다.

> 프로그래밍 유경험자들에게:
>
> C 기반의 언어들과 대조적으로, 스몰토크에서는 인덱스가 0 이 아닌 1 로 시작합니다.

at:put:

배열의 특정 위치에 객체를 두기 위해 `at:put:` 메시지를 사용합니다.

```
anArray at: 1 put: 'Toothbrush'
```

```
'Toothbrush'
```

인터렉티브 모드에서 객체의 이름을 입력하는 것만으로 배열 객체의 값을 확인할 수 있습니다.

```
anArray
```

```
('Toothbrush' nil nil nil nil nil nil nil nil nil )
```

이제 보시다시피 첫번째 객체는 String 클래스의 인스턴스 'Toothbrush' 입니다. 이제 soap 를 배열에 입력해보겠습니다.

```
anArray at: 2 put: 'Soap'
```

```
'Soap'
```

```
anArray
```

```
('Toothbrush' 'Soap' nil nil nil nil nil nil nil nil )
```

includes:

특정 객체를 배열이 갖고 있는지 `includes:` 메시지를 전달하여 확인할 수 있습니다.

```
anArray includes: 'Soap'
```

```
true
```

```
anArray includes: 'Toothpaste'
```

```
false
```

reverse

Array 객체가 순서를 갖는 집합이라고 말한 것을 기억하십시오. 이 순서를 다루는 방법이 있습니다. 그 중 그 순서를 반대로 만드는 것이 `reverse` 메시지입니다.

```
anArray reverse
```

```
(nil nil nil nil nil nil nil nil 'Soap' 'Toothbrush' )
```

Set

Set 은 Array 와 비슷하지만, 순서없이 객체를 담아둔다는 점과 갯수의 한계를 미리 정의하지 않는다는 점이 다릅니다. GNU 스몰토크에서의 Set 은 수학에서 말하는 집합과 비슷한 것입니다. 따라서 수학에서의 집합과 같이 생각하시면 됩니다.

new

new 메시지로 Set 클래스의 인스턴스를 생성할 수 있습니다. 그러니 이 경우 메시지에 다른 인자가 필요없습니다.

```
| aSet |

aSet := Set new
```

```
Set ()
```

초기화 한 것을 우리에게 보여주지만 어떤 객체도 포함하지 않았다는 것을 보여줍니다.

add:

Set 에 add: 메시지를 이용하여 객체를 추가할 수 있습니다.

```
aSet add: 'Toothbrush'
```

```
'Toothbrush'
```

```
aSet
```

```
Set ('Toothbrush' )
```

이제 우리는 'Toothbrush' 라는 값을 갖는 string 객체를 넣어두었습니다. 이제 다른 String 객체를 추가해봅시다.

```
aSet add: 'Soap'
```

```
'Soap'
```

```
aSet
```

```
Set ('Soap' 'Toothbrush' )
```

GST 가 'Toothbrush' 앞에 객체를 둔 것을 확인할 수 있을 것입니다. 실제로는 이전 객체의 앞에 둔 것이 아니라, 그냥 임의로 집어넣은 것입니다. 'Soap'와 'Toothbrush'의 객체 위치는 예측하지 못하며, 다른 말로 Set 안에선 확실한 위치를 알 수 없다는 것입니다.

> 질문:
>
> 왜 Set 클래스가 reverse 메시지를 갖고 있지 않은지 설명할 수 있습니까?

아마도 다음 예제를 보면 이해하기 쉬울 것입니다.

```
aSet add: 'Toothpaste'
```

```
'Toothpaste'
```

```
aSet
```

```
Set ('Soap' 'Toothpaste' 'Toothbrush' )
```

이제 우리가 추가한 마지막 객체가 두 객체 사이에 있습니다. 하지만, 또한 이것도 역시 실제로는 순서가 없습니다.

Set 객체 안에서는 순서를 알 수 없다는 것이 흥미로운 점입니다. 두 번 이상 같은 객체

를 Set 객체에 집어넣으려 해도, 같은 객체는 들어가지 않습니다.

```
aSet add: 'Toothbrush'
```
```
'Toothbrush'
```

```
aSet
```
```
Set ('Soap' 'Toothpaste' 'Toothbrush' )
```

GST 에게는 객체가 Set 안에 있는지 없는지가 중요합니다. 앞으로 차차 보시게 되겠지만, 이것은 수학의 집합과 매우 흡사한 점입니다.

remove:

Set 에 remove: 메시지를 보내어 지정한 객체를 제거할 수 있습니다.

```
aSet remove: 'Toothpaste'
```
```
'Toothpaste'
```

`remove:` 메시지가 Set 객체를 반환하지 않고 삭제한 객체를 보여준다는 것을 주의하십시오.

```
aSet
```
```
Set ('Soap' 'Toothbrush' )
```

보신 것처럼, Set 의 항은 인덱스가 없기 때문에, 인덱스가 아닌 항을 가지고서 제거합니다.

집합에서 각 항을 제거하였다고 하여서, 그것이 시스템에서 객체가 삭제되는 것을 의미하지는 않습니다. 다음의 예를 보십시오.

```
| anElement |

anElement := 'Perfume'
```
```
'Perfume'
```

별도의 변수에 String 객체를 지정하였습니다. 이제 집합에 추가하도록 하겠습니다.

```
aSet add: anElement
```
```
'Perfume'
```

```
aSet
```
```
Set ('Soap' 'Toothbrush' 'Perfume' )
```

이제 제거해보겠습니다.

```
aSet remove: anElement
```

```
'Perfume'
```

```
aSet
```

```
Set ('Soap' 'Toothbrush' )
```

이제 Set 으로부터 항목을 제거하였습니다. 아직 변수가 살아있는지 확인해보겠습니다.

```
anElement
```

```
'Perfume'
```

보다시피 여전히 잘 존재합니다. 우리가 Set 에 보낸 메시지 remove: 메시지는 String 객체와 이를 가리키는 변수에 영향을 미치지 않았습니다.

Dictionary

이제 보여드릴 마지막 클래스는 Dictionary 라고 합니다. Dictionary 는 Array 와 Set 과 같이 여러 개의 객체를 묶는 기능을 제공하고 있습니다.

Dictionary 는 각 데이터를 관련된 인덱스 키와 연결하여 자료를 유지한다는 것이 Array 와 같습니다. 그러나 Dictionary 의 키는 정수가 아니어도 된다는 점이 다릅니다. 자료에 대한 키로서 어떠한 객체도 정의할 수 있습니다.

Dictionary 는 Set 과 같이 순서가 없다는 것, 그리고 객체의 갯수를 제한하지 않는다는 점이 같습니다.

new

Set 과 같이 new 메시지를 사용할 수 있습니다.

```
| aDictionary |

aDictionary := Dictionary new
```

```
Dictionary (
)
```

초기에 Dictionary 객체가 아무 객체도 포함하지 않았다는 것을 볼 수 있습니다.

at:put:

at:put: 메시지를 사용하여 사전에 객체를 추가할 수 있습니다.

```
aDictionary at: 'Canol' put: 'Gokel'
```

```
'Gokel'
```

```
aDictionary
```

```
Dictionary (
```

```
        'Canol'->'Gokel'
)
```

이제 사전은 Canol 로 이름붙인 문자열과 연결된 Gokel 이라는 문자열을 갖고 있습니다. 이제 다른 String 객체를 추가해봅시다.

```
aDictionary at: 'Paolo' put: 'Bonzini'
```

```
'Bonzini'
```

```
aDictionary
```

```
Dictionary (
        'Canol'->'Gokel'
        'Paolo'->'Bonzini'
)
```

keys

keys 메시지로 Dictionary 안의 모든 키를 얻을 수 있습니다. GST 는 포함하고 있는 모든 키를 Set 객체로 반환할 것입니다.

```
aDictionary keys
```

```
Set ('Canol' 'Paolo' )
```

removeKey:

Dictionary 로부터 항목을 제거하기 위해 removeKey: 메시지를 사용할 수 있습니다.

```
aDictionary removeKey: 'Canol'
```

```
'Gokel'
```

```
aDictionary
```

```
Dictionary (
        'Paolo'->'Bonzini'
)
```

removeKey: 메시지는 Set 의 remove: 메시지와 같이 제거한 객체를 반환합니다.

연습문제

1. 객체, 메시지, 클래스는 무엇입니까?
2. 변수란 무엇입니까? 변수는 왜 필요합니까?
3. 메시지의 종류에는 어떤 것이 있습니까? 일반 클래스와 그 사용법 절에서 봐왔던 메시지들 가운데, 메시지의 종류마다 2 개의 예를 꼽을 수 있습니까?
4. GNU 스몰토크는 수를 어떤 형식으로 다룹니까? 각각의 형식에 대한 예를 들어주십시오.

5. 사용자로부터 수를 입력 받아, 그 수의 세제곱을 구하는 프로그램을 작성하십시오.

6. 사용자로부터 공백으로 구분 된 두 수를 입력 받아, 두 수의 평균을 출력하는 프로그램을 작성하십시오.

   (힌트 : 3 4 와 같이 사용자로부터 String 객체로 두 수를 얻을 것입니다. 문자열에 tokenize: aString 메시지를 전달하여, 두 수를 구분하십시오. tokenize: 메시지는 인자로 주어진 하나의 문자열 안에서 토큰(token)이라 부르는 조각으로 구분하여, 구성하고 있던 모든 토큰을 Array 객체 형태로 반환합니다.)

7. 1 장에서 봤던 여러 개념들의 정의문을 Dictionary 에 담는 프로그램을 작성하십시오. 프로그램은 사용자로부터 용어를 입력 받아 그것의 정의문을 표시해야 합니다.

8. Array 와 Dictionary 의 차이점은 무엇입니까? Dictionary 는 이름도 멋진데다가, 기능도 더 많은데, 왜 굳이 Array 을 써야할까요?

지식인은 단순한 것을 어렵게 말하는 사람이고
예술가는 어려운 것을 단순하게 말하는 사람이다.

찰스 부코스키 (Charles Bukowski)

제 4 장 예외 흐름 제어

여태까지 봐왔던 프로그램은 입력한 코드 순서대로 한 줄씩 수행하였습니다. 그러나 실제 삶은 그렇지 않습니다. 우리는 성장해야하고, 어떤 결정을 내리기도 해야합니다. 프로그램은 현실 세계 문제의 모방체이기 때문에 판단력을 요하는 순간이 옵니다. 결정은 코드가 비순차적으로 실행되게 만듭니다. 어떤 코드는 판단에 의해 실행될 것이며, 또 어떤 코드는 전혀 실행되지 않기도 할 것입니다. 예를 들어, 날씨가 춥다면 외투를 입어야 할 것이고, 춥지 않다면 입지 않아야 할 것입니다.

결정을 내리는 것이 프로그램의 실행을 제어하는 것의 전부가 아닙니다. 반복도 있습니다. 반복은 실생활에선 없는 것이지만 프로그래밍 언어가 존재하는 가장 큰 이유입니다. 예를 들어, 여러분이 어떤 작업을 수 천번 하길 원한다고 합시다. 가장 먼저 떠오르는 문제는 수학문제입니다. 1 에서 1000 억까지의 수 중에서 씨수를 모두 찾고 싶다고 합시다. 건강을 생각해봐도, 종이와 연필로 구한다는 것은 확실히 좋지 않습니다. 대신 수가 소수인지 아닌지를 알아내는 알고리즘을 개발하여서, 1 부터 1000 억까지의 숫자에 대입해보면 됩니다. 이 문제의 답은 요즘 컴퓨터를 사용하면, 아무런 계산 실수 없이 몇 밀리초만에 풀어버릴 수 있습니다.

따라서 제어를 하는 방식으로 선택과 반복, 두 가지 형식이 있습니다. *선택 제어* *selective controlling* 는 조건에 따라 코드 중 일부분을 선택하여 수행합니다. 그리고 *반복 제어* *repetitive controlling* 는 코드 중 일부를 계속 반복한다는 것입니다.

GNU 스몰토크로 어떻게 선택제어와 반복제어를 하는지 알아보기 전에, 제어 메시지들과 함께 아주아주 많이 사용할 블록 block 이라는 것에 대해 공부해 봅시다.

블록

블록은 나중에 실행될 수식 표현을 담고 있는 객체입니다. 그저 대괄호 안에 작성한 코드일 뿐입니다. 블록의 예입니다.

```
['Hello World!' printNl.
(3 + 7) printNl]
```

기억해야 할 점은 작성한 블록도 객체이며, 블록 안의 코드도 바로 실행하는 것이 아니라는 점입니다. 블록에 `value` 라고 이름붙인 메시지를 전달해보면 다음과 같습니다.

```
['Hello World!' printNl.
(3 + 7) printNl] value
```

```
'Hello World!'
10
10
```

객체 10 이 두 번 출력되었는데, 블록의 소스코드 중 마지막 수식표현의 값으로 객체 10 을 반환하였기 때문입니다. 다시 말하지만, 이것은 터미널에서 인터렉티브 모드로 코드를 수행하였을 때의 얘기입니다.

제어 표현에서 보겠지만, 블록은 매우 유용한 객체 구조입니다. 제어 표현으로 들어가기 전에, 블록의 요소 중 하나인, 블록의 인자에 대해 말해보겠습니다.

때때로 블록은 안의 코드에서 처리해야 할 추가 자료를 필요로 하는 경우가 있습니다. 그 때에는 추가적인 자료를 하나 이상의 *블록 인자*로 만들어서 전달하면 됩니다. 보통 인자를 갖는 블록의 구조는 다음과 같습니다.

```
[:blockArgument1 :blockArgument2 | block-expression-1. block-
expression-2]
```

블록 인자들은 블록의 시작 부분에 위치하며, 모든 인자들은 앞에 콜론을 붙이도록 되어 있습니다. 블록의 인자부분은 파이프 문자로 블록의 주요 내용과 구분합니다.

앞으로 대부분 이러한 종류의 블록을 특수한 메시지와 함께 사용할 것입니다. 그러나 여러분이 이러한 블록을 직접 사용하길 원한다면, 여러분은 인자 갯수만큼의 value: 메시지를 포함한 선택자를 갖춘 메시지를 보내야 할 것입니다. 예를 들어, 블록이 세 개의 인자를 필요로 한다면, 여러분은 `value:value:value:` 선택자를 사용해야 할 것입니다.

예제로 이 주제를 마무리하겠습니다.

```
"blocks.st"
"A program which involves a block with an argument."


| greetings |

greetings := [:platesOfCornFlakes | 'I have eaten ',
platesOfCornFlakes printString, ' plates of corn flakes this
morning!'].

('Hello ma! ', (greetings value: 3)) printNl.
```

```
'Hello ma! I have eaten 3 plates of corn flakes this morning!'
```

여기서 가장 먼저 앞으로 만들게 될 블록을 가리킬 greetings 라는 변수를 만들었습니다. 그리고 인자가 있는 블록을 생성하여 변수에 지정했습니다. 주의할 점은 표현식 안에서 인자를 쓸 때에는 인자 앞에 콜론을 붙이지 않았다는 것입니다. 인자를 선언할 때에만 콜론을 쓰고, 이후에는 생략합니다. 마지막으로 greeting value: 3 이라는 표현과 함께 이 블록을 실행하였습니다. 우선순위를 고려하여 괄호로 묶어주는 것을 잊지 마십시오.

이제 제어 표현으로 진행할 준비가 되었습니다.

선택 제어

ifTrue:

첫 선택 제어 메시지는 `ifTrue:` 입니다. 일반적인 구조는 다음과 같습니다.

```
anObject ifTrue: [block-expression-1. block-expression-2]
```

Boolean 객체에 이 메시지가 전달되면, 객체가 true 일 때, 코드 블록 내의 표현을 실행하며, false 일 때에는 무시합니다.

예를 들어보겠습니다.

```
| ourVariable |

ourVariable := true.

ourVariable ifTrue: [
      'Our variable is true.' printNl.
]
```

```
'Our variable is true.'
```

`ourVariable`이라고 부르는 변수를 생성하고, true 객체를 지정합니다. 그 다음 `ifTrue:` 메시지를 보냅니다. `ourVariable`이 true로 지정되어 있기 때문에, 코드블록을 수행합니다.

> 질문:
>
> `ourVariable`을 false 객체로 할당하고 같은 코드를 수행하여 보십시오.

ifFalse:

`ifFalse:` 메시지는 그것이 false 객체에게 보내졌을 때 코드 블록을 수행하는 것을 제외하면 `ifTrue:`와 같습니다.

ifTrue:ifFalse:

이제 여러분의 객체가 참일 때 수행할 코드와 거짓일 때 수행할 코드에 대해 써 볼 시간입니다. 이 메시지는 여러분들에게 쉬운 방법을 제공합니다. 일반 구조를 보면 다음과 같습니다.

```
anObject ifTrue: [block-expression-1. block-expression-2] ifFalse:
[block-expression-3. block-expression-4]
```

물론 더욱 보기 좋게 다음과 같이 쓴다고 하여도 문제 없습니다.

```
an-object
      ifTrue: [
            the-code-block-to-execute
      ] ifFalse: [
            the-code-block-to-execute
      ]
```

이제 `ifTrue:ifFalse:` 메시지를 사용하는 예를 하나 들어서 선택 제어 메시지를 끝낼까 합니다.

```
| ourVariable |

ourVariable := false.

ourVariable ifTrue: [
      'Our variable is true.' printNl.
] ifFalse: [
      'Our variable is false.' printNl.
]
```

```
'Our variable is false.'
```

반복 제어

프로그래밍 언어나 기계를 사용해야 하는 중요한 이유 중 하나를 배울 시간입니다. 기계는 지치거나 실수를 하지 않기 때문에, 여러분이 수백만번 똑같은 일을 하도록 시킬 수도 있으며, 대부분 밀리초 단위로 아주 빠르게 일할 것입니다. 오늘날 컴퓨터의 도움이 없는 분야는 생각할 수도 없습니다. 예를 들어, 군사목적 하드웨어는 빠르고 복잡한 계산을 제한 시간 안에 정확하게 수행해야 합니다. 혹은 천문학 같은 과학분야의 계산은 종이와 연필로 할 수 없는 어마어마한 계산을 수행해야 합니다.

스몰토크는 이러한 목적에서 예외적인 언어가 아닙니다. 이제부터 가장 흔하게 쓰이는 반복 표현을 생성하기 위한 메시지들을 볼 것입니다.

whileTrue:

이 메시지의 일반 구조는 다음과 같습니다.

```
aBlock whileTrue: anotherBlock
```

이 메시지는 블록 객체에 보내도록 설계되었습니다. 객체에 메시지를 전달하면, 블록 객체를 평가하여, true 일 경우 인자로 전달한 블록을 수행합니다. 이러한 점에서 직접 Boolean 객체를 사용하지 않는 것과, 블록을 전달한다는 것을 제외하면 ifTrue:메시지와 아주 흡사합니다. 그러나 이제부터가 확연히 다른 부분입니다. 만약 메시지를 전달받는 객체가 true 로 평가되면 인자 블록을 수행하며, 이후에 이 블록을 한 번 더 제어하는 것입니다. 만약 그때까지도 블록의 실행값이 true 라면, 인자 블록을 한 번 더 수행합니다. 이 순환은 메시지를 받는 블록의 실행값이 false 가 될 때까지 계속됩니다.

while 반복을 *결정되지 않은 반복* *indefinite loop* 이라고 부르는데, 얼마나 많은 횟수를 반복할지 정하지 않기 때문입니다. 만약 횟수를 안다면, 여러분의 설계를 다시 확인해보기 바랍니다. 왜냐하면 앞으로 배우게 될 루프 기술들이 그런 경우에 더 적합하기 때문입니다.

이 메시지를 더 잘 이해하기에 좋은 예제를 보여드리겠습니다.

```
"average.st"
"A program which evaluates the sum of the numbers entered to
demonstrate the whileTrue: message."

| sum enteredIntegers lastEnteredInteger |
```

```
sum := 0.
enteredIntegers := 0.

[ lastEnteredInteger ~= -1 ] whileTrue: [
      Transcript cr; show: 'Please enter a number. To exit the
program enter -1: '.
      lastEnteredInteger := stdin nextLine asInteger.

      sum := sum + lastEnteredInteger.
      enteredIntegers := enteredIntegers + 1.


      Transcript show: 'The average of the numbers entered so far
is: ', (sum / enteredIntegers) printString; cr.
]
```

```
Please enter a number. To exit the program enter -1: 3
The average of the numbers entered so far is: 3

Please enter a number. To exit the program enter -1: 4
The average of the numbers entered so far is: 7/2

Please enter a number. To exit the program enter -1: 3432
The average of the numbers entered so far is: 3439/3

Please enter a number. To exit the program enter -1: -1
The average of the numbers entered so far is: 1719/2
```

이 예제는 사용자가 숫자를 입력해주어야 합니다. 그리고 입력했던 값들의 평균값을 계산하여 출력하여 줍니다. 프로그램을 멈추는 방법은 -1 을 입력하는 것입니다. `lastEnteredInteger` 라고 부르는 변수에 수를 입력하면, 매 회마다 -1 인지 아닌지 판단을 합니다.

```
[ lastEnteredInteger ~= -1 ] whileTrue: [
      ...
]
```

연산자 ~= 는 좌변의 객체가 우변의 객체와 다른지를 확인하는 연산자입니다.

다른 입력으로 진행하는 동안 GNU 스몰토크 가상머신은 분수 형태로 평균값을 보여줍니다.

to:do:

이 메시지의 일반 구조는 다음과 같습니다.

```
aNumber to: anotherNumber do: aBlock
```

이 구조는 숫자 aNumber 로부터 시작하여 anotherNumber 가 될 때까지 aBlock 객체를 실행합니다. 반복할 때마다 수는 늘어납니다. aBlock 객체는 블록 인자를 포함하고 있으

며, aBlock 표현 안에서 현재 인덱스를 사용할 수 있습니다.

`whileTrue:` 메시지는 보통 결정되지 않은 반복의 경우에 사용합니다. 대조적으로 `to:do:`는 반복횟수가 얼마나 필요할지 아는 경우에 사용합니다. 이것이 *결정된 반복*definite loop 이라고 불리는 이유입니다.

> 프로그래머 유경험자들에게:
>
> 한정된 반복을 위해 C 기반 언어에서의 for 반복문 대신 스몰토크에서는 이 메시지가 사용된 점을 짐작할 수 있습니다.

이제 예제를 통해 메시지 구조를 확인하여 보겠습니다.

```
"5_lines.st"
"A program which prints 5 lines to demonstrate the usage of to:do:
message."

1 to: 5 do: [:x |
      Transcript show: 'This is the ', x printString, '. line.';
cr.
]
```

```
This is the 1. line.
This is the 2. line.
This is the 3. line.
This is the 4. line.
This is the 5. line.
```

이 프로그램은 각각의 인덱스 값을 표시하며 다섯 개의 행을 출력합니다. 블록 인자를 x 를 각 반복의 인덱스 값으로 둔 것에 주의하십시오. 만약 블록 객체에 대한 인자를 쓰지 않았다면, 에러가 났을 것입니다.

to:by:do:

때때로 `to:do:` 메시지의 인덱스를 하나씩 증가시키고 싶지 않을 때가 있습니다. 이러한 목적으로 대신 호출하는 메시지가 `to:by:do:` 입니다. 일반 구조는 다음과 같습니다.

```
aNumber to: anotherNumber by: step do: aBlock
```

aNumber 객체의 값이 anotherNumber 에 다다를 때까지 step 으로 정의한 수만큼 증가할 것입니다.예를 들어 위의 예제에서 인덱스를 2 씩 증가하여 실행하도록 해보겠습니다. 대신 이번에는 10 까지 증가하도록 하여서, 반복이 빨리 끝나지 않도록 하겠습니다.

```
"tobydo.st"
"A program to demonstrate the usage of to:by:do: message."

1 to: 10 by: 2 do: [:x |
      Transcript show: 'This is the ', x printString, '. line.';
cr.
```

```
]
```

```
This is the 1. line.
This is the 3. line.
This is the 5. line.
This is the 7. line.
This is the 9. line.
```

출력을 보면 어떤 영향이 있었는지를 알게될 겁니다. 이 메시지는 반대 방향으로 줄어드는 데에도 쓸 수 있습니다.

```
"tobydo_backwards.st"
"A program to demonstrate the backward capability of to:by:do:
message."

5 to: 1 by: -1 do: [:x |
      Transcript show: 'Oh my god! I''m counting backwards! This is
the ', x printString, '. line!'; cr.
]
```

```
1. Oh my god! I'm counting backwards! This is the 5. line!
    Oh my god! I'm counting backwards! This is the 4. line!
    Oh my god! I'm counting backwards! This is the 3. line!
    Oh my god! I'm counting backwards! This is the 2. line!
    Oh my god! I'm counting backwards! This is the 1. line!
```

이번 장은 이것으로 마칩니다. 다음 장에서는 객체지향 프로그래밍에 대한 개념과 클래스를 만드는 법을 배워보면서 우리의 여행을 지속하겠습니다.

연습문제

1. 제어 메시지의 최종 목적은 무엇입니까? 어떤 종류의 제어 메시지들이 있습니까? 각각의 종류에 대한 예를 들어보십시오.
2. 블록은 무엇입니까? 왜 블록이 필요합니까?
3. 결정된 반복과 결정이 안 된 반복은 무엇입니까? 각각의 반복에 대한 메시지를 하나 씩 예를 들어 보십시오.
4. 2 장 연습문제에서 다이아몬드를 출력하는 프로그램을 이번 장에서 공부한 개념을 사용하여 다시 작성하여 보십시오. 얼마나 많은 슬래시 문자를 사용할지

도 입력받아서, 더 크게도, 작게도 만들어 보십시오.

```
   /\
  /  \
 /    \
 \    /
  \  /
   \/
```

5. 주어진 수가 소수인지 아닌지 판별하여 결과를 알려주는 프로그램을 작성하십시오. -1 을 입력할 때까지 프로그램을 계속 실행하도록 만드십시오.

6. 99-bottles-of-beer.net 웹 사이트로부터 영감을 받은 문제입니다. http://99-bottles-of-beer.net/lyrics.html 을 찾아서 99 Bottles of Beer 노래의 가사를 출력하는 프로그램을 작성하십시오.

7. 각각 결정된 반복과 결정이 안 된 반복 방식을 이용하여 1 부터 10 까지의 수를 표시하는 프로그램을 만드십시오. 어떤 방법이 이 작업을 하는 데에 적합합니까? 왜 그렇습니까?

8. *입력한 숫자들의 평균을 구하는 프로그램을 결정된 반복과 결정이 안 된 반복 방식으로 작성하여 보십시오. 프로그램을 종료하기 원할 때에는* `finish` *라고 입력하면 끝나도록 만들어야 합니다.*

9. 이제 2 장 4 번 문항에서 얘기한 한 줄 짜리 프로그램을 작성할 수 있습니까?

나의 뇌: 그것은 내가 두 번째로 가장 좋아하는 기관(organ)이다..

우디 알렌 (Woody Allen)

제 5 장 객체, 메시지, 클래스 : 2 부

캡슐화

여러분 핸드폰의 전자회로가 어떻게 작동하는지 궁금하진 않았나요? 그걸 알아야 하지는 않나요? 아마도 그런 경우는 거의 없을 것입니다. 핸드폰을 사용하기 위해서 어떻게 만들었는지, 모두 알아야 할 필요는 없습니다. 그냥 메뉴얼을 읽는 것만으로도 충분히 사용할 수 있습니다. 따라서, 우리는 모든 자세한 내용, 과학 법칙, 사용할 물체의 모든 속성을 알 수도 없지만, 알 필요도 없습니다. 객체 지향 프로그래밍은 -- 위와 같은 실제 생활의 방식을 응용프로그램에 접목한-- *캡슐화* *encapsulation* 라는 것을 합니다.

스몰토크에서는 객체의 내부 상태를 직접 바꾸도록 허용하지 않습니다. 객체는 오로지 객체끼리 메시지를 전달하며 소통할 뿐입니다. 따라서, 스몰토크는 불필요한 정보를 캡슐화합니다.

게다가 캡슐화는, 여러분에게 실세계 상태에서 사고하는 것을 허용하며, 몇몇 종류의 *정보 은닉* *information hiding* 을 통해, 여러분이 작성할 프로그램을 더욱 견고하게 하고 재사용 가능하도록 도움을 줍니다. 이 말은 여러분이 객체 내부 구현 내용을 숨기고, 다른 사람들에게 객체와 소통하기 위한 *프로토콜* *protocol* 만을 제공하여서, 차후에 구현 내용이 바뀌더라도, 객체를 사용하는 프로그래머들이 기존 프로토콜 만으로 영향을 받지 않고, 계속 사용할 수 있다는 것을 의미합니다.

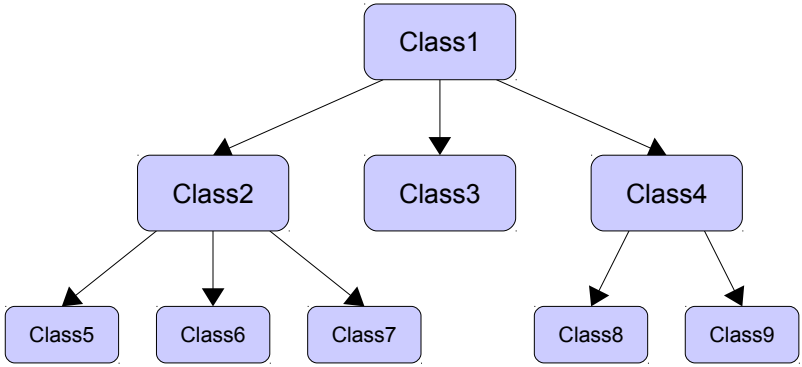
예를 들어, 여러분이 이차방정식의 제곱근을 구하는 Calculator 클래스를 생성했습니다. 방정식의 정보를 보내주기만 하고, 제곱근 값을 반환받습니다. 이 클래스는 어쩌면 제곱근을 구하는데 몇 시간이 걸리는 비효율적인 알고리즘을 사용할 수도 있습니다. 그래도 없는 것보다는 나으니까, 여러분은 친구들에게 코드를 배포하였습니다. (그들을 클라이언트라고 부를 수 있겠습니다.) 아마도, 배포 한 후에, 여러분이 혁신적인 접근을 발견하여, 계산시간을 몇 초 단위로 줄이게 되었습니다. 알고리즘이 변하였지만, 프로토콜은 변할 필요가 없습니다. 왜냐하면, 클라이언트들은 계산기에 똑같은 정보를 전달하고, 클래스가 방정식의 해를 구해주기만을 원하기 때문입니다. 어떻게 방정식을 풀었는가는 그들의 관심사 밖입니다. 따라서 여러분은 캡슐화의 이점을 얻을 수 있고, 여러분의 새로운 코드를 클라이언트들에게 배포합니다. 클라이언트는 Calculator 클래스를 교체하기만 하면 되고, 그들의 프로그램을 변경할 필요는 없습니다.

상속

인간은 무언가를 분류하는 것을 좋아합니다. 분류를 하면 짜임새가 생기게 되고, 짜임새 있는 것은 다루기가 쉽습니다. 비슷한 것들은 모여서 그룹(혹은 분류)를 만들고, 그 그룹의 특성들 중에는 다른 그룹에는 없는 것이 있을 것입니다. 우리는 유사성에 따라 어떤 그룹들을 같이 모을 수도 있습니다. 이러한 과정은 모든 다른 그룹의 조상이라는 하나의 큰 그룹으로 도달할 때까지 계속됩니다. 이것은 실제로 삶의 시뮬레이션 그 자체입니다.

위에서 말한 요소는 객체이며, 그룹은 클래스입니다. 요소에 대하여 가장 일반적으로 사용하는 표현은 무엇일까요? 그 답은 객체입니다. 우리는 모든 것을 객체라고 부를 수

있으며, 이들 객체로부터 특별한 종류의 객체를 이끌어 내어 우리는 자동차, 집, 동물 등과 같은 이름을 붙입니다. 이들 객체로부터 특별한 종류의 객체를 이끌어 내어 우리는 자동차, 집, 동물 등과 같은 이름을 붙입니다. 따라서 우리가 자동차, 집, 동물을 부를 때, 실제로는 클래스에 관하여 말하고 있는 것입니다.



이 다이어그램에서 각 노드는 클래스를 표현합니다. 어떤 클래스는 다른 클래스로부터 도출하였습니다. 가지들은 다른 클래스로부터 도출하였다는 것을 표현하기 위한 것입니다. 예를 들어, Class2, Class3 그리고 Class4 는 모두 Class1 으로부터 도출하였습니다. 또한 Class5, Class6, Class7 은 Class2 로부터 도출하였습니다. 그리고 마지막으로 Class8 과 Class9 는 Class4 로부터 도출하였습니다. 클래스간의 이러한 관계를 나타내기 위해 두가지 정의를 만들 것입니다.

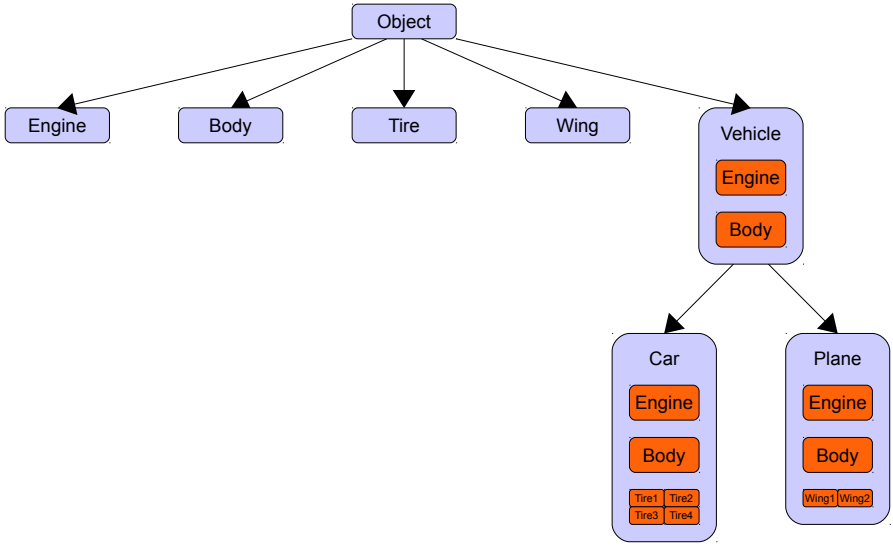
다른 클래스로부터 도출해 낸 클래스들을 도출해 내는데 사용한 클래스의 *하위클래스* *subclass* 라고 부릅니다. 예를 들어, Class2, Class3 와 Class4 는 모두 Class1 의 하위 클래스들입니다. 또한 Class5, Class6 그리고 Class7 은 모두 Class2 의 하위클래스입니다.

모든 클래스는 자신의 하위클래스의 *상위클래스* *superclass* 입니다. 예를 들어, Class1 은 Class2, Class3, Class4 의 상위 클래스입니다.

도출한 클래스(하위 클래스)들은 그들의 조상(상위클래스)의 모든 속성과 동작을 상속받습니다.

위의 다이어그램에서 어떤 클래스들은 하위 클래스를 가지고 있으며, 모든 클래스는 Class1 을 제외하고 상위클래스를 가지고 있다. 클래스들은 하나 이상의 하위클래스들을 가질 수는 있어도, 여러 개의 상위클래스를 가질 수는 없습니다.

아래의 예제 그림은 객체 개념의 관점에서 자동차와 비행기를 어떻게 생성하는지 보여주고 있습니다.



이 그림에서 각 노드는 클래스를 나타냅니다. 8 개의 클래스가 있으며, 각각 Object, Engine, Body, Wing, Tire, Vehicle, Car, Plane 이라고 이름붙였습니다. Engine, Body, Wing 그리고 Vehicle 은 모두 Object 의 모든 하위 클래스입니다. Car 와 Plane 은 Vehicle 의 하위클래스입니다. 하나의 클래스는 다른 클래스들의 인스턴스로 구성할 것입니다. Vehicle, Car, Plane 클래스 안의 그러한 예제를 봅시다. Vehicle 클래스는 Engine 과 Body 클래스의 인스턴스를 갖고 있습니다. Car 와 Plane 클래스들은 Vehicle 의 하위클래스이기 때문에 상위클래스인 Vehicle 의 모든 속성과 행동을 상속받습니다. Engine 과 Body 객체들을 상속받은 후, 두 객체를 구분하기 위해 Car 클래스에는 4 개의 Tire 객체들을 추가하고, Plane 클래스에는 두 개의 Wing 객체를 추가합니다.

상속에 관한 혼란을 피하기 위해 만들 두 가지 개념이 있습니다. 그것들은 아마도 객체들간의 두 가지 관계일 것입니다. 먼저 객체는 다른 객체로부터 도출할 것입니다. 이것을 is-a 관계라고 부릅니다. 예를 들어, Car 는 Vehicle 입니다 (Car is a Vehicle). 다음으로 객체는 다른 객체들로 이루어져 있습니다. 이러한 관계를 has-a 관계라고 부릅니다. 예를 들어, Vehicle 은 Body 를 가지고 있습니다 (Vehicle has a Body).

상속 개념은 is-a 관계와 관련이 있으며, has-a 와는 관련이 없습니다.

다중 상속과 단일 상속

어떤 프로그래밍 언어는 프로그래머가 하나 이상의 클래스로부터 속성과 동작을 상속

받는 클래스를 만들 수 있도록 허용합니다. 따라서, 어떤 클래스들은 하나 이상의 상위클래스를 갖습니다. 이를 *다중 상속* *multiple inheritance* 이라고 합니다. 다른 상속 형식은 오로지 하나의 상위클래스만을 허용합니다. 이를 *단일 상속* *single inheritance* 이라고 합니다.

아주 이전에 주어졌던 예제는 단일 상속이었으며, 앞으로 나올 예제들도 단일 상속일 것입니다. 왜냐하면 스몰토크는 클래스를 다루기 위해 단일 상속만을 사용하기 때문입니다. 이 부분은 객체 지향 설계에서 다중 상속이 필요한지 아니면 단순히 상속을 더욱 복잡하게 할지 아직도 논란의 여지가 많습니다.

다형성 (Polymorphism)

다른 종류의 객체들로 가득찬 세상에서 어떤 객체들은 다른 종류의 객체, 즉, 다른 클래스의 인스턴스이지만, 같은 메시지에 응답하는 경우가 있습니다. 이것은 어떤 프로그래밍 언어가 다른 클래스에서 같은 선택자로 메소드를 정의할 수 있도록 하는 이유입니다. 혹은 상의 클래스에서 물려받은 선택자의 행동을 하위 클래스에서 변경 가능합니다. 이것을 *다형성* *polymorphism* 이라고 합니다.

> 주의:
>
> 하위클래스에서 상위클래스로부터 상속 받은 메소드를 바꾸는 것을 오버라이딩(overriding)이라고 합니다.

이제 이 개념에 대해서 고전적이지만 아주 자명한 예제를 들어보겠습니다. Animal 이라는 클래스와 두 하위클래스 Dog 과 Cat 을 생각해보십시오. Animal 클래스에서는 speak 메소드를 구현할 것입니다. 왜냐하면 동물은 어떤 종류의 잡음을 내는 것으로 다른 생물과 소통하기 때문입니다 (개는 소리를 내는 것보다 깨무는 것으로 소통하기도 합니다만). 그러나 이 메시지를 보냈을 때, 개들은 짖을 것이며, 고양이는 야옹거릴 것입니다. 따라서 다른 방식으로 같은 메시지에 응답한다는 것입니다.

나만의 클래스 만들기

GNU 스몰토크는 내장된 클래스만 수백개이지만, 온 세상은 셀 수 없을 정도로 많은 클래스를 가지고 있기 때문에, GNU 스몰토크에서는 여러분이 필요한 클래스를 만들 수 있습니다. 우리가 만든 클래스를 때로는 *사용자 정의 클래스* *custom classes* 라고 부르겠습니다.

먼저, 클래스는 어떤 것들으로 만들어져 있는지 다뤄보겠습니다. 클래스들은 객체의 청사진과 같습니다. 따라서 객체가 가지는 모든 것을 명확하게 서술해야 합니다. 예를 들어, 클래스는 객체의 속성을 정의하기 위한 인스턴스 변수를 가지고 있습니다. 클래스는 어떤 특정 메시지에 대해 객체가 응답할 수 있도록 메소드 정의를 가지고 있습니다. 클래스는 위의 설명이 다가 아닙니다. *클래스 변수* *class variable* 라는 것이 있습니다. 클래스 변수는 클래스의 상태에 대한 정보를 담고 있습니다. 또한 클래스만 실행 가능한 클래스 메소드도 있습니다. *클래스 메소드* *class method* 는 클래스만 실행시킬 수 있지, 인스턴스에서는 실행시킬 수는 없습니다.

클래스 생성을 위한 문법은 다음과 같습니다.

```
SuperclassName subclass: SubclassName [
```

```
        | instanceVariable1 instanceVariable2 |

        classVariable1 := anObject.
        classVariable2 := anotherObject.

        <comment: 'Comment to describe our class'>

        SubclassName class >> aClassMethod: aParameter [
                "Comment to describe this class method"
                <category: 'Category of this class method'>

                | localVariable1 localVariable2 |

                ...

                ^objectToReturn
        ]

        anInstanceMethod [
                "Comment to describe this instance method"
                <category: 'Category of this instance method'>

                | localVariable1 localVariable2 |

                ...

                ^objectToReturn
        ]
]
```

위의 가상 코드는 클래스를 어떻게 만들고, 클래스와 인스턴스 변수들을 어떻게 추가하고, 클래스와 인스턴스 메소드를 어떻게 추가하는지 보여주고 있습니다. 소스코드를 한 줄씩 설명하기 전에 몇몇 일반적인 개념에 대해 말하겠습니다. 위 소스코드의 대부분은 부가적인 것입니다. 예를 들어 클래스 변수나 메소드를 꼭 가지고 있을 필요는 없습니다. 혹은 주석을 달아둘 필요도 없으며, 메소드의 분류를 정의할 필요도 없습니다. 그러나 이것들은 소프트웨어 엔지니어링의 좋은 습관입니다. 그리고 항상 주석을 남기고, 메소드의 분류를 정의하기를 권장합니다.

또한 위의 코드에서 관심있는 부분은 공백의 사용입니다. 몇몇 행은 탭이나 빈칸으로 *들여쓰기 indent* 가 되어있지만, 왜 그렇게 되어 있겠습니까? 이는 인간의 눈으로 코드를 매끄럽게 보기 위한 것입니다. 또한 공백을 어떻게 사용할지에 대한 규칙은 없습니다. 대부분 어떤 새로운 구조의 본문을 들여쓰기합니다. 예를 들어, 클래스의 본문을 작성할 때, 다른 내용과 비교하여 한 단계 안으로 들여쓰기 합니다. 여기서 한 단계는 프로그래머마다 다릅니다. 들여쓰기 하는 데에 여러개의 빈 칸을 선택하거나 탭을 넣을 수도 있지만, 중요한 점은 프로그램을 통틀어 일관성을 갖는 것이 혼란스럽게 보이지 않습니다.

들여쓰기 외에도, 사용하는 대부분의 공백들이 프로그래머마다 다릅니다. 여러 단어

마다 앞뒤로 빈칸을 두거나 두지 않는 것을 선택할 수 있습니다. 만약 잘못 사용하면(모호함을 초래할 때) 인터프리터는 여러분에게 오류를 전달할 것입니다. 예를 들어, 여러분은 변수 이름과 키워드 선택자를 연결하여 쓸 수 없습니다. 왜냐하면 인터프리터는 변수명이 어디에서 끝나는지, 키워드 선택자가 어디에서 시작하는지 알 수 없기 때문입니다. 따라서 적어도 빈 칸 하나를 두어야만 합니다.

위의 코드를 한 줄씩 보도록 하겠습니다. 먼저 한 일은 아래의 코드 내용으로 클래스를 정의하는 것입니다.

```
SuperclassName subclass: SubclassName [
        ...
]
```

코드에서 세 개의 점은 거기에 더 많은 수식 표현이 있다는 것을 의미합니다 (앞으로 늘 이렇게 표시하겠습니다). 진짜로 중요한 부분을 강조하기 위해 짧게 줄인 것입니다. 실제 코드에 점 세개를 두면 안됩니다. 위의 부분에서 SuperclassName 클래스에게 SubclassName 이라는 하위 클래스를 만들도록 지시한 것입니다. 상위클래스의 속성과 동작을 상속하기 위해 지정한 것이기 때문에 클래스를 다시 만들지 않아도 됩니다. 그런 다음, 클래스 변수, 인스턴스 변수, 클래스 메소드, 인스턴스 메소드를 지정하기 위해 대괄호를 열었습니다.

위의 대괄호가 열린 부분을 클래스의 *헤더* *header* 라고 부릅니다. 대괄호 사이의 부분을 클래스의 *본문* *body* 이라고 부릅니다.

파이프 사이에 둔 이름들은 클래스의 인스턴스 변수들입니다.

```
        | instanceVariable1 instanceVariable2 |
```

인스턴스 변수 이름들은 공백으로 구분합니다. 두 인스턴스 변수는 instanceVariable1 과 instanceVariable2 로 이름붙였습니다.

이제 지정 연산자를 사용하여 클래스 변수를 정의하겠습니다.

```
        classVariable1 := anObject.
        classVariable2 := anotherObject.
```

:= 메시지는 전달받는 객체를, 인자 객체를 참조하는 객체로 만들어주는 역할을 합니다. 따라서 이제부터 classVariable1 을 사용할 때, 인터프리터는 실제로 anObject 객체를 참조한다는 것으로 이해할 것입니다. 인스턴스 변수를 변경하는 것에 대해서도 앞으로 이 지정 연산자 메시지를 사용할 것입니다.

처음 변수에 관하여 언급하는 것을 *변수 선언*이라고 말합니다. 변수를 사용하기 전에 먼저 선언해야만 합니다.

다음 표현은 클래스에 주석을 추가합니다.

```
        <comment: 'Comment to describe our class'>
```

주석은 문서 목적으로만 쓰이며, 사용하는 IDE 에서는 클래스나 메소드가 어떤 일을

하는지 설명하기 위해서 나타납니다. 따라서 시간을 들여 힘들게 코드를 보지 않고도 코드가 무엇을 하는지 생각을 얻을 수 있습니다. 주석은 프로그램 실행에 영향을 미치지 않기 때문에, 생략할 수 있습니다.

클래스와 인스턴스 메소드를 정의할 시간입니다. 먼저 클래스 메소드를 만들겠습니다.

```
SubclassName class >> aClassMethod: aParameter [
        "Comment to describe this class method"
        <category: 'Category-of-this-class-method'>

        | localVariable1 localVariable2 |

        ...

        ^objectToReturn
]
```

여기 헤더부분과 본문부분이 있습니다. 클래스 선언의 헤더는 다음과 같습니다.

```
SubclassName class >> aClassMethod: aParameter
```

이 헤더의 SubclassName class >> 부분은 클래스의 인스턴스가 아닌, 클래스에 대한 메소드를 생성할 것을 알려주고 있습니다. 만약 이 부분을 생략하면 메소드는 이 클래스의 인스턴스의 것이 됩니다. 이후 이 부분을 메소드의 메시지라고 합니다. 이 메시지의 선택자는 aClassMethod:입니다. 파라미터는 aParameter 라고 이름붙였습니다. *파라미터*parameter 들은 메소드의 헤더에서 사용할 때 인자이름으로 주어진 이름입니다. 따라서 이 단어는 실제로 인자와 다를바 없습니다. 그리고 때때로 교환해서 사용합니다. 메소드 안의 파라미터 이름을, 메소드에 전달하기 위한 인자를 참조하는 방법으로 사용합니다.

선택자를지정한 후에 메소드의 본문이라 불리는 메소드 내용을 정의하기 위해 대괄호를 열었습니다.

```
[
        "Comment to describe this class method"
        <category: 'Category of this class method'>

        | localVariable1 localVariable2 |

        ...

        ^objectToReturn
]
```

메소드의 내용은 주석, 범위, 몇몇의 지역 변수와, 점 세개로 표시한 실행할 표현식으로 구성할 수 있습니다. 주석을 쓴 후에

```
        "Comment to describe this class method"
```

다음과 같이 이 메소드의 분류를 지정할 수 있습니다.

```
        <category: 'Category of this class method'>
```

이것은 부가적인 것입니다. 따라서 우리는 주석을 쓰거나 메소드에 대한 분류를 지정할 필요는 없습니다.

메소드의 본문은 중요한 부분입니다. 먼저 지역 변수를 선언합니다.

```
        | localVariable1 localVariable2 |
```

이는 인스턴스 변수를 선언하는 것과 같습니다. 지역 변수는 메소드에 한하여 지역적이며, 메소드 본문 바깥에서는 사용할 수 없습니다.

다음으로 객체 이름 앞에 ^ 문자를 붙여서 호출자에게 되돌려줄 변수로 `objectToReturn` 객체를 반환합니다.

```
        ^objectToReturn
```

이 부분은 반환 표현이며, 메시지가 실행된 결과 값이기도 합니다. 예를 들어, 2 + 3 을 수행할 때, 그러한 표현을 통해 5 라는 값을 반환할 것입니다. 반환 표현을 사용하여 값을 반환하는 것 또한, 두가지 이유에서 부가적인 것입니다. 첫번째로, 모든 메소드가 응답 객체를 필요로 하는 것은 아닙니다. 마치 표준 출력에 몇몇 문자를 표시하는 것과 같습니다. 두번째로 메소드는 여러분이 반환 기호를 표시하지 않으면, 마지막 수식표현을 반환합니다. 그러나 반환함을 정확하게 명시하는 것은 좋은 프로그래밍 습관이므로, 소스코드를 읽는 사람들이 명확하게 알 수 있도록 반환값을 표현해주는 것이 좋습니다.

마지막으로 말할 것은 인스턴스 메소드입니다.

```
anInstanceMethod [
        "Comment to describe this instance method"
        <category: 'Category of this instance method'>

        | localVariable1 localVariable2 |

        ...

        ^objectToReturn
]
```

이 코드 부분에 대해선 자세히 이야기하지 않겠습니다. 왜냐하면 인스턴스 메소드를 만드는 헤더부분에 클래스 이름이 없다는 것을 제외하고는 거의 클래스 메소드와 흡사하기 때문입니다. 따라서 이 메소드는 모든 인스턴스에 공통적으로 존재하게 될 것입니다.

클래스로부터 객체 생성하기

클래스를 만든 후, 아래 코드의 형식처럼 사용하여 객체를 생성할 수 있습니다.

```
ourObjectName := SubclassName new
```

우리는 SubclassName 으로부터 객체를 생성하기 위해 new 메시지를 보냈습니다. 그러나, 잠시만! 우리는 new 메소드를 정의한 적이 없습니다. 그렇다면 어떻게 new 메소드를 사용할 수 있을 까요? 이것은 상속의 개념과 관련이 있습니다. 모든 클래스, 객체들은 기본으로 정의한 메소드들을 가지고 있는데, new 메소드 또한 그 중 하나입니다. 이 메소드는 클래스로부터 객체를 만드는 방법으로 제공하는데, 상속을 하기 때문에 매번 다시 정의할 필요는 없습니다.

`ourObjectName` 은 새로 생성한 객체를 참조할 이름입니다. 이는 *변수* *variable* 라고 불리기도 합니다. 이제 프로그램에 `ourObjectName` 을 쓸 때, 인터프리터는 우리가 말하는 객체가 무엇인지 알 것입니다. 예를 들어, 객체에서 사용하기 위해 `anInstanceMethod` 를 선언하였던 것을 기억하실 것입니다. 우린 다음과 같은 표현을 사용하여 객체에 이 메시지를 전달할 수 있습니다.

```
ourObjectName anInstanceMethod
```

먼저 변수의 이름과 전달할 메시지의 이름을 씁니다. 이 표현은 `objectToReturn` 을 반환할 것입니다.

인스턴스 메소드를 어떻게 사용하는지는 배웠지만, 클래스 메소드는 어떻게 사용하는지 배운 적은 없습니다. 아래의 문장을 실행하면 에러가 날 것입니다.

```
ourObjectName aClassMethod
```

`aClassMethod` 는 인스턴스가 아니라 클래스를 위한 것입니다. 다음과 같이 클래스를 위한 메소드를 수행할 수 있습니다.

```
SubclassName aClassMethod
```

혹은 객체의 클래스에 도달하기 위해 (new 에서 그랬던 것처럼) 상속받은 메소드 class 로부터 도움을 얻을 수 있습니다. 그리고 다음과 같이 메시지를 전달할 수 있습니다.

```
ourObjectName class aClassMethod
```

여기서 연쇄 메시지 개념을 사용하였습니다. 메시지가 왼쪽에서부터 오른쪽으로 판단되기 때문에, 우리의 표현은 다음의 표현으로 먼저 전환합니다.

```
SubclassName aClassMethod
```

예제

이제 실제 예제를 이용하여 우리가 배운 개념을 모두 설명해보겠습니다. 이 예제에서 우리는 동물들에 대한 클래스를 만들 것입니다. Animal 이란 이름의 클래스는 모든 동물에 대한 정보를 가지고 있으며, 특별한 클래스 Dog 과 Cat 은 각 종류의 특별한 정보를

갖고 Animal 객체로부터 도출하였습니다.

```
"animal.st"
"객체지향 개념을 보기 위해 동물 클래스를 만드는 프로그램."

Object subclass: Animal [
      | name |

      animalNumber := 0.

      <comment: '동물을 정의하는 클래스'>

      Animal class >> setAnimalNumber: number [
              "동물이 몇 마리 있는지 지정하는 클래스 메소드"
              <category: 'accessing'>

              animalNumber := number.

              ^animalNumber
      ]

      Animal class >> getAnimalNumber [
              "동물이 몇 마리 있는지 가져오는 클래스 메소드"
              <category: 'accessing'>

              ^animalNumber
      ]

      setName: newName [
              "동물의 이름을 정하는 인스턴스 메소드"
              <category: 'accessing'>

              name := newName.
      ]

      getName [
              "동물의 이름을 가져오는 인스턴스 메소드"
              <category: 'accessing'>

              ^name
      ]
]

Animal subclass: Dog [
      <comment: '개 클래스'>
```

```
	makeNoise [
		"개의 울음소리를 가져오는 인스턴스 메소드"
		<category: 'accessing'>

		'Woof!' printNl.
	]
]

Animal subclass: Cat [
	<comment: 'A cat class.'>

	makeNoise [
		"고양이의 울음소리를 가져오는 인스턴스 메소드"
		<category: 'accessing'>

		'Miaow!' printNl.
	]
]

dog := Dog new.
Animal setAnimalNumber: 1.
dog setName: 'Karabash'.
dog getName printNl.
dog makeNoise.

cat := Cat new.
Animal setAnimalNumber: 2.
cat setName: 'Minnosh'.
cat getName printNl.
cat makeNoise.
```

```
'Karabash'
'Woof!'
'Minnosh'
'Miaow!'
2
```

이제 이 프로그램을 한 줄씩 설명해보겠습니다. 먼저 모든 클래스의 조상격인 특수 객체 Object 를 상속하여 클래스부터 만듭니다. 여기에는 new 와 같이 미리 정의된 유용한 메소드도 따라옵니다.

```
Object subclass: Animal [
	...
]
```

다음으로 name 이라 이름붙인 인스턴스 변수와 animalNumber 라는 이름의 클래스 변수를 선언합니다. 동물에 대한 이름과 동물의 개체수를 클래스 변수에 담아둘 것입니다.

```
        | name |

        animalNumber := 0.
```

클래스에 주석을 쓰는 것은 문서를 남기기 위한 목적입니다.

```
<comment: 'A class for defining animals.'>
```

이제 클래스의 동작을 제공할 메소드를 정의해봅시다. 먼저 동물의 개체수를 지정할 수 있는 클래스 메소드를 만듭니다.

```
        Animal class >> setAnimalNumber: number [
                ...
        ]
```

그리고 동물의 개체수를 얻기 위한 메소드도 만듭니다.

```
        Animal class >> getAnimalNumber [
                ...
        ]
```

프로그래밍 하는 동안 set-get 과 같은 종류의 쌍을 자주 볼 것입니다. 나중에 그것들을 *획득자* *getter* 와 *지정자* *setter* , 혹은 *접근자* *accessor* 라고 부릅니다.

다음으로 동물의 이름을 지정하고 얻기 위한 접근자를 만듭시다.

```
        setName: newName [
                ...
        ]

        getName [
                ...
        ]
```

이제 첫번째 클래스 작성을 완료하였습니다. 이 클래스로부터 Dog 과 Cat, 두 개의 클래스들을 도출해낼 것입니다.

```
Animal subclass: Dog [
        ...
]

Animal subclass: Cat [
        ...
]
```

각 클래스는 Animal 클래스로부터 상속하였기 때문에, 이미 setName:과 getName 메소드를 가지고 있습니다. 일부러 다시 작성할 필요가 없습니다. 대신 각 클래스에 makeNoise 메소드를 추가 했습니다. 이러한 방식을 선택한 이유는 각각의 클래스가 고유의 소리를 내기 때문입니다.

코드의 나머지 부분은 시험을 할 목적으로 있는 것입니다. 두 개의 객체를 만들었고, 적절한 메시지를 전달하는 것으로 메소드를 시험합니다.

```
dog := Dog new.
Animal setAnimalNumber: 1.
dog setName: 'Karabash'.
dog getName printNl.
dog makeNoise.

cat := Cat new.
Animal setAnimalNumber: 2.
cat setName: 'Minnosh'.
cat getName printNl.
cat makeNoise.
```

먼저 Dog 객체를 만들고 동물 개체수를 1 로 설정한 다음 개의 이름을 'Karabash'로 지정합니다. 그리고 개의 이름을 출력하고 짖게 만듭니다.

다음으로 Cat 객체를 만들어서 Dog 과 같은 과정을 적용합니다. 이번에는 이름을 'Minnosh'로 정하고, 동물 개체수를 2 로 설정합니다. 왜냐하면 동물이 두 마리이기 때문입니다.

> 프로그래밍 유경험자들에게:
>
> 실제로 현재와 같은 상속 트리의 디자인이나 동물 개체수를 직접 설정하는 것은 실제 응용프로그램에서는 물론 나쁜 형태입니다만, 예제를 간단히 유지하기 위해 사용하였습니다.

마지막 명령문은 Animal 클래스 안에 유지하고 있는 동물 개체수를 출력하는 것입니다.

```
Animal getAnimalNumber printNl.
```

이 예제에서 접근자 메소드들을 상속하는 것을 보았습니다. 접근자 메소드는 클래스 변수와 인스턴스 변수로 구성된 객체의 내부와 소통할 수 있도록 해주었습니다. 우리는 `makeNoise` 를 통해서 다형성(메시지를 받은 객체가 자신의 클래스에 따라 다르게 행동하는 것)을 살펴 보았습니다. 그리고 Animal 클래스로부터 상속된 모든 클래스에 `setName:`과 `getName` 메소드가 있다는 것을 확인함으로써 상속 개념도 볼 수 있었습니다.

클래스 확장과 변경

프로그램에서 수의 세제곱을 많이 계산한다고 가정해봅시다. Number 클래스가 수신자 객체의 세제곱수를 답하는 `cubed` 라는 메소드를 가지고 있는 것이 아주 유용하지 않겠습니까? 프로그래머는 자주 그런 식으로 클래스를 수정할 필요가 있으며, GNU 스몰토크는 그러한 방법을 제공합니다. 내장된 클래스도 프로그래머가 만든 클래스도 수정할 수 있습니다.

따라서 여러분은 클래스 내의 새로운 메소드를 추가하거나 기존 메소드를 수정할 수

있습니다. 클래스를 수정하기 위한 일반 문법은 다음과 같습니다.

```
className extend [
      | newInstanceVariable1 newInstanceVariable2 |

      methodSelector [
              ...
      ]
]
```

인스턴수 변수들을 클래스에 추가할 수 있습니다.

만약 같은 선택자의 메소드가 이미 정의되었다면, 마지막 메소드가 나머지 모두를 대체하며 새롭게 구현할 것입니다.

만약 같은 선택자의 메소드가 존재하지 않는다면, 추가할 것입니다.

클래스 자체를 고치길 원한다면, 여러분은 예시처럼 `className` 다음에 `class` 를 추가합니다.

```
className class extend [
      | newClassVariable1 newClassVariable2 |

      methodSelector [
              ...
      ]
]
```

이번에는 클래스변수와 클래스메소드를 만든 것이라는 점에 유의하세요.

이제 Human 클래스를 만들고 수정하여봅시다. Human 클래스의 핵심 버전은 같은 공통 속성을 갖을 것이며, 기본 메시지에 답변할 수 있을 것입니다.

```
"human.st"
"Human 클래스의 핵심버전"

Object subclass: Human [
      | name age |

      setName: aName [
              name := aName.
      ]

      getName [
              ^name
      ]

      setAge: anAge [
              age := anAge.
```

```
	]

	getAge [
		^age
	]

	introduceYourself [
		Transcript show: 'Hello, my name is ', name, ' and
I''m ', age printString, ' years old.'; cr.
	]

	> aHuman [
		^age > aHuman getAge
	]

	< aHuman [
		^age < aHuman getAge
	]

	= aHuman [
		^age = aHuman getAge
	]
]

| me myBrother |

me := Human new.
me setName: 'Canol Gökel'.
me setAge: 24.

myBrother := Human new.
myBrother setName: 'Gürol Gökel'.
myBrother setAge: 27.

me introduceYourself.
myBrother introduceYourself.

(me < myBrother) printNl.
```

```
Hello, my name is Canol Gökel and I'm 24 years old.
Hello, my name is Gürol Gökel and I'm 27 years old.
true
```

우리 Human 클래스는 아주 간단합니다. name 과 age 로 이름 붙인 두 개의 인스턴스 변수가 있습니다. 접근자(`setName:`, `getName` 등)를 정의한 다음에 `introduceYourself` 라고 이름 붙인 메소드를 정의하였습니다. Human 객체가 이 메시지를 받으면 객체가 가지고 있는 어떤 정보를 스스로 말하여 자기 자신을 소개합니다. 두 Human 의 나이를 비교하는 메소드 또한 작성하였습니다. 만약 두 Human 객체간의

비교하길 원한다면, 여러분은 이항 메시지인 >, < 나 =을 사용할 수 있습니다. 수신자 객체의 나이를 접근할 때는 age 라는 인스턴스 변수를 사용하였고, 인자 객체의 나이를 접근할 때는 getAge 라는 메시지를 보냈다는 차이점을 주의깊게 보아 주십시오. 비교 메소드는 이미 Integer 객체에서 정의하였으며, 더 이상 깊게 들어가지 않겠습니다.

이 구현은 아주 간단합니다. 이 클래스에 좀더 세부적인 것을 추가하고 싶다고 칩시다. 위의 코드를 직접 고쳐도 되겠지만, 그렇게 할 수 없을 때도 있을 겁니다. 예를 들어, 하나 이상의 프로그램이 코드를 공유하고 있고, 그 중 하나의 객체만 확장된 버전이 필요할 때가 있습니다. 그런 경우에는 원본 코드를 유지하는 것이 좋은 생각입니다. 자, 그럼 새로운 시험용 프로그램에서 Human 클래스를 확장해 봅시다. 표시된 부분에는 이전의 Human 클래스의 코드를 복사해서 붙여넣으십시오.

```
"human_extended.st"
"Human 클래스를 확장한 프로그램"

... -> 여기에 옛 Human 클래스의 코드를 붙여 넣으세요.

Human extend [
      | occupation experience |

      getOccupation [
            ^occupation
      ]

      setOccupation: anOccupation [
            occupation := anOccupation.
      ]

      getExperience [
            ^experience
      ]

      setExperience: anExperience [
            experience := anExperience.
      ]

      introduceYourself [
            Transcript show: 'Hello, my name is ', name, ' and
I''m ', age printString, ' years old. I am ', occupation, '.'; cr.
      ]

      > aHuman [
            ^experience > aHuman getExperience
      ]
```

```
        < aHuman [
                ^experience < aHuman getExperience
        ]

        = aHuman [
                ^experience = aHuman getExperience
        ]
]

| me myFriend |

me := Human new.
me setName: 'Canol Gökel'.
me setAge: 24.
me setOccupation: 'an Engineer'.
me setExperience: 1.

myFriend := Human new.
myFriend setName: 'İsmail Arslan'.
myFriend setAge: 23.
myFriend setOccupation: 'an Engineer'.
myFriend setExperience: 3.

me introduceYourself.
myFriend introduceYourself.

(me > myFriend) printNl.
```

```
Hello, my name is Canol Gökel and I'm 24 years old. I am an
Engineer.
Hello, my name is İsmail Arslan and I'm 23 years old. I am an
Engineer.
false
```

우리는 위 코드에서 많은 작업을 했습니다. 하나씩 살펴 봅시다. 맨 처음에 한 일은 occupation 과 experience 라는 두 개의 인스턴스 변수를 추가한 것입니다. occupation 은 직업을 뜻하며, experience 는 경력을 뜻합니다.

```
        | occupation experience |
```

그런 다음, 우리는 새 인스턴스 변수들을 위한 접근자 메소드들을 작성하였습니다.

```
        getOccupation [
                ^occupation
        ]

        setOccupation: anOccupation [
                occupation := anOccupation.
        ]
```

```
        getExperience [
                ^experience
        ]

        setExperience: anExperience [
                experience := anExperience.
        ]
```

이들 메소드들은 전에 작성된 적이 없기 때문에, 기존 클래스에 추가 되었습니다. 그런 다음, 우리는 몇몇 메소드를 재작성하였습니다.

```
        introduceYourself [
                Transcript show: 'Hello, my name is ', name, ' and
I''m ', age printString, ' years old. I am ', occupation, '.'; cr.
        ]

        > aHuman [
                ^experience > aHuman getExperience
        ]

        < aHuman [
                ^experience < aHuman getExperience
        ]

        = aHuman [
                ^experience = aHuman getExperience
        ]
```

이 메소드들은 오래된 구현을 오버라이딩(overridding) 했습니다. 새 소개 메소드는 Human 의 직업도 포함하고 있습니다. 비교 메소드도 변경하여서 이번에는 Human 객체의 경력에 따라 비교가 이뤄질 것입니다. 이런 변경은 마치 인사과에서 그들의 프로그램에서 Human 클래스를 쓰기 위해 변경한 것 같군요.

self 와 super

이제 GNU 스몰토크의 새로운 2 개의 키워드를 배울 시간입니다. 이제 볼 두 키워드의 용도는 같습니다. 클래스를 정의하는 동안 메시지를 받는 객체를 언급하는 것입니다. 차이점은 참조한 객체 상에서 메소드를 호출하려 할 때 나타날 것입니다.

self

먼저 볼 키워드는 `self` 입니다. 클래스 선언에서 이 키워드를 쓸 때, 메시지를 받는 객체로 처리합니다. 예를 들어, 기본 Human 클래스를 아래와 같이 정의할 수 있습니다.

```
"human_self.st"
"The second version of the Human class written using self
keywords."
```

```
Object subclass: Human [
	| name age |

	setName: aName [
		name := aName.
	]

	getName [
		^name
	]

	setAge: anAge [
		age := anAge.
	]

	getAge [
		^age
	]

	introduceYourself [
		Transcript show: 'Hello, my name is ', self getName, '
and I''m ', self getAge printString, ' years old.'; cr.
	]

	> aHuman [
		^self getAge > aHuman getAge
	]

	< aHuman [
		^self getAge < aHuman getAge
	]

	= aHuman [
		^self getAge = aHuman getAge
	]
]

| me myBrother |

me := Human new.
me setName: 'Canol Gökel'.
me setAge: 24.

myBrother := Human new.
myBrother setName: 'Gürol Gökel'.
myBrother setAge: 27.
```

```
me introduceYourself.
myBrother introduceYourself.

(me < myBrother) printNl.
```

```
Hello, my name is Canol Gökel and I'm 24 years old. I am an
Engineer.
Hello, my name is İsmail Arslan and I'm 23 years old. I am an
Engineer.
false
```

차이점은 객체의 속성을 직접 사용하는 대신 `self` 키워드를 사용하여 객체의 속성에 접근한다는 것입니다. 예를 들어, `self getName` 은 "메시지를 받을 객체에게 `getName` 메시지를 보내라"라는 뜻입니다.

`self` 가 메시지를 받으면, 그 객체의 클래스에서부터 그것의 모든 상위클래스까지 메시지와 일치하는 메소드를 찾을 때까지 검색합니다. 이것이 `self` 과 `super` 의 차이이니까, (`super` 의 설명이 나올 때까지) 잘 기억해 두세요.

위의 예제처럼 속성에 직접 접근 가능하더라도 접근자를 사용하는 것을 권장합니다. 이것이 객체지향의 캡슐화 속성이며, 좋은 프로그래밍 실천법입니다. 왜냐하면, 실생활에서 객체의 내부가 바뀌면, 무슨 일이 일어날지 알 수가 없기 때문입니다. 예를 들어, Molecule 객체가 있다고 가정합시다. 원자를 추가하려 할 때, 결합 방식, 분자 모형등의 여러가지 molecule 속성이 변화할 것입니다. 단순하게 `oxygen := oxygen +1` 라고 해서는 안 됩니다. 예상할 수 없는 모든 세세한 것들을 다룰 수 있도록 `addOxygenAtom` 과 같은 방법을 사용해야 한다는 것입니다.

물론 어떤 때에는 직접 접근해야 할 때가 있겠지만(접근자들이 대표적인 사례), 가급적이면 추상화시키도록 노력하세요.

super

직직접 클래스를 작성하여 메소드를 재정의할 때, 상위클래스의 메소드 정의를 가져와야할 경우가 있습니다. 상위클래스의 메소드를 완전히 바꾸기보다는 추가적인 동작만을 넣고 싶을 때가 그런 경우에 해당하겠습니다. 이 예제는 손목시계 (Watch) 클래스와, 이 클래스로부터 또 다른 클래스 방수되는 손목시계(water resistant watches) 를 만드는 예제입니다.

```
"watches.st"
"키워드 super 를 사용하는 방법을 보여주기 위하여 손목시계 클래스를 정의하는 프
로그램"

Object subclass: Watch [
    | style chronometerCapability |

    <comment: '보통 손목시계를 정의하는 클래스'>

    setStyle: theStyle [
```

```
        "손목시계의 양식을 결정한다."

        style := theStyle.
    ]

    getStyle [
        "손목시계의 양식을 가져온다."

        ^style
    ]

    setChronometerCapability: theChronometerCapability [
        "고정밀 시간 측정 능력을 명시하는 메소드"

        chronometerCapability := theChronometerCapability.
    ]

    getChronometerCapability [
        "시계가 고정밀 시간 측정 능력을 가지고 있는지 판별하는 메소드"

        ^chronometerCapability.
    ]

    listYourFeatures [
        "손목시계의 기능을 출력하는 메소드"

        Transcript show: 'Style: ', self getStyle; cr.
        Transcript show: 'Chronometer capabilities: ', self
getChronometerCapability printString; cr.
    ]
]

Watch subclass: WaterResistantWatch [
    | resistanceDepth |

    <comment: '방수 시계를 정의하는 클래스'>

    setResistanceDepth: aDepth [
        "방수가 보장되는 수심(깊이)를 결정하는 메소드"

        resistanceDepth := aDepth.
    ]

    getResistanceDepth [
        "방수가 보장되는 수심(깊이)를 가져오는 메소드"
```

```
        ^resistanceDepth.
    ]

    listYourFeatures [
        "방수시계의 기능을 출력하는 메소드"

        super listYourFeatures.

        Transcript show: 'Resistance depth: ', self
getResistanceDepth printString; cr.
    ]
]

| watch1 watch2 |

watch1 := Watch new.
watch1 setStyle: 'Analog'.
watch1 setChronometerCapability: true.

watch2 := WaterResistantWatch new.
watch2 setStyle: 'Digital'.
watch2 setChronometerCapability: false.
watch2 setResistanceDepth: 30.

Transcript show: 'The features of the 1st watch:'; cr.
Transcript show: '------------------------------'; cr.
watch1 listYourFeatures.

Transcript cr.

Transcript show: 'The features of the 2nd watch:'; cr.
Transcript show: '------------------------------'; cr.
watch2 listYourFeatures.
```

```
The features of the 1st watch:
------------------------------
Style: Analog
Chronometer capabilities: true

The features of the 2nd watch:
------------------------------
Style: Digital
Chronometer capabilities: false
Resistance depth: 30
```

이 프로그램은 먼저 일반 시계를 표현하는 Watch 클래스를 정의합니다. 아날로그인지, 디지털인지, 초정밀 시간 측정이 가능한지 등의 여부를 알 수 있는 `style` 이라는 속성과 `chronometerCapability` 속성도 지정할 수 있습니다. 이 속성에 대한 접근자 메

소드를 정의하고 나서, 표준 출력에 요소들을 출력하기 위한 다른 `listYourFeatures` 메소드에 대해서도 정의합니다. 지금까지 특별한 내용은 없습니다.

Watch 클래스를 정의한 후, 이 클래스로부터 방수시계를 표현하는 WaterResistantWatch 클래스를 만듭니다. 이 객체에 방수 한계 심도를 지정할 새 객체 `resistanceDepth`를 추가합니다. 이제부터가 묘미입니다. 우리는 상위 클래스로부터 `listYourFeatures` 메소드를 오버라이딩하여 메소드 시작 부분에 다음과 같은 한 줄을 써두었습니다.

```
    super listYourFeatures.
```

이 구문은 `listYourFeature` 메소드를 자기 자신에게 보내는 것이지만, `self` 키워드가 아닌 `super` 키워드에 보냈기 때문에 메소드의 정의를 탐색할 때, 자신의 클래스가 아니라 자신의 상위클래스(우리의 예제에서는 Watch 클래스)부터 탐색한다는 점이 다릅니다. 따라서 WaterResistantWatch 클래스의 `listYourFeatures` 메소드 대신 Watch 클래스의 `listYourFeatures` 메소드를 호출합니다.

이렇게 하는 이유는 Watch 클래스가 이미 WaterResistantWatch 클래스의 속성을 많이 가지고 있고, 손목시계의 기능들을 나열하는 기능을 가지고 있기 때문입니다. 새 클래스 메소드에 기존 `listYourFeatures` 내용을 복사해서 붙여넣을 수도 있고 방금처럼 상위클래스를 내부에서 정의하는 메소드를 호출할 수도 있습니다. 프로그래머는 후자를 사용합니다. 소스코드를 다시 작성하는 작업도 줄이고, 소스코드를 리펙토링 할 때, 하나의 소스코드만을 수정해도 되기 때문입니다. `super` 키워드를 사용했기 때문에, Watch 클래스에 새 속성이 추가된다고 해도 우리의 코드(WaterResistantWatch)는 영향받지 않습니다. `super` 키워드를 쓰지 않는다면, Water 클래스의 코드가 바뀔 때마다, 그 부분을 WaterResistantWatch 에 복사해야 합니다.

여기에 `self` 키워드를 사용하는 것은 잘못된 방법입니다. 왜냐하면 재귀적으로 자기 자신을 계속 호출하여서 무한 루프를 유발하기 때문입니다. 무한재귀를 차치하더라도, 우리가 원하는 코드는 상위클래스에 있지 우리 클래스에 있지는 않기 때문에, 그 문장은 우리가 원하는 것을 해주지도 못합니다.

프로그램의 나머지 부분은 Watch 클래스와 WaterResistantWatch 클래스의 시계를 정의하고, 각각의 속성을 설정한 후, 나열합니다. 의도한 대로, 두 번째 시계는 상위 클래스로부터 속성을 상속받아 같이 표시하며, 추가한 특수 속성 `resistanceDepth` 도 같이 출력합니다.

연습문제

1. 다형성(polymorphism), 상속, 캡슐화에 대하여 몇 문장으로 간단히 서술하시오.
2. 더 크다는 의미가 아닌 더 낫다는 의미로 기호 > 에 대하여 생각해보십시오. Human 클래스를 상속받은 Man 클래스를 만들어서, 추가로 money 와 handsomeness 인스턴스 변수를 포함하십시오. 우리 모두 알다시피 외면보다는 내면의 아름다움이 중요합니다. 그래서 Woman 클래스에 추가로 honesty 와 generosity 인스턴스 변수를

만들어주십시오. 모든 인스턴스 변수는 10 점 만점의 정수를 사용합니다.

Man 클래스에 Man 객체와 Man 객체를 비교하는 >를 구현하되, 메시지를 받는 객체의 money 와 handsomeness 의 합이 인자로 넘어온 객체의 money 와 handsomeness 의 합보다 크면 true 를 답하게 작성하세요. 단, 두 객체 중에 이름이 "Canol Gökel"인 것이 있으면, 그 객체가 언제나 이기게 만들어야 합니다.

Women 클래스도 비슷한 방법으로 > 메소드를 구현하십시오. honesty 와 generosity 의 포인트를 더하여 각각 true 와 false 로 반환하십시오. 이번에는 예외상황이 없습니다. 여성들은 모두 똑같으니까요...

3. Number 클래스를 cubed 메시지를 받으면, 객체의 값의 세제곱 값을 출력하도록 확장하십시오.

4. self 와 super 키워드의 목적을 설명하고, 각각의 차이점을 설명하십시오.

내게 가장 생산적인 날 중 하나는,
1000 라인 이상의 코드를 버렸던 날이다.

캐네시 톰슨 (Kenneth Thompson)

제 6 장 다음으로 할 것들

어떻게 스스로 발전할까요?

비록 스몰토크가 문법적으로는 작은 언어지만, 이것으로 할 수 없는 일이 거의 없고, 그 양이 방대한 내장된 클래스와 거기에 포함된 메소드들이 여러분을 도와줄 것입니다. GNU 스몰토크의 문법적인 모든 부분을 거의 다 다루었으며, 또한 자주 사용하는 클래스들도 설명하였습니다. 아직 탐험해야 할 많은 클래스들이 여러분 앞에 기다리고 있습니다. 이들을 살펴보기 위한 가장 효과적인 방법은 연습을 하는 것입니다. 프로그래밍 세계에서 연습이란 쉽게 말해 프로그램을 작성해보는 것입니다. 이미 여러분은 진행하고 싶은 프로젝트를 맘 속에 품고 있을 것입니다. 하지만 만약 그런 것이 없다면 GNU 스몰토크로 풀어볼 수 있는 흥미로운 문제들이 많이 있습니다. 이 장 이후 '더 읽을 만한 것들' 부분에서 몇 개의 웹사이트를 소개하겠습니다.

또 다른 중요한 점은 다른 스몰토크 프로그래머들과 만나고, 새로운 스몰토크 뉴스를 접하여 여러분이 혼자라고 느끼지 않는 것입니다. 이 장의 끝에는 몇몇 웹사이트, 메일링 리스트, 블로그 주소를 써두었습니다.

이 책을 읽어서 이론적인 부분을 배운 여러분들이라면, 이러한 것들이 여러분의 실전 능력을 배양시켜줄 것입니다. 이 다음 부분부터 여러분의 이론적인 지식을 개선하여 줄 것들, 또한 실전 프로그래밍에서 필요한 필수 내용들을 다룬 책들도 있습니다. 여기에는 일반적인 고급 프로그래밍 개념과 스몰토크 언어와 관련한 고급 주제가 빠져있지만, 알아두면 스몰토크가 다른 관점에서 아름다운 언어라는 것을 알 수 있을 것입니다. 이 또한 우리가 왜 다음 절의 책들을 읽어야 하는지에 대한 이유가 될 것입니다.

더 읽을 만한 것들

Smalltalk-80: The Language and its Implementation by Adele Goldberg and David Robson

이 책은 Smalltalk-80 프로그래밍 언어를 기술한 공식 책입니다. 또한 이 언어를 어떻게 구현하였는가도 다룹니다. 이 책은 Blue Book 으로 알려져 있습니다. 이 책을 읽으면 스몰토크 프로그래밍 언어에 관한 세세한 사항을 알 수 있습니다.

GNU Smalltalk Documentation - http://smalltalk.gnu.org/documentation

GNU 스몰토크에 관한 더 자세한 정보를 찾아 볼 수 있습니다. 여기엔 라이브러리 레퍼런스와 사용자 메뉴얼이 있습니다.

유용한 사이트들

GNU Smalltalk - http://smalltalk.gnu.org

공식 GNU 스몰토크 사이트입니다. GNU 스몰토크의 최신 버전, 뉴스, 문서를 얻을 수 있습니다. 또한 스몰토크 개발자와 사이트 회원들이 쓰는 블로그도 있습니다.

Smalltalk.org - http://www.smalltalk.org

스몰토크와 관련한 모든 종류의 정보가 있는 거대한 사이트입니다.

Stephane Ducasse :: Free Online Books - http://stephane.ducasse.free.fr/FreeBooks.html

스몰토크 언어와 관련하여 자유롭게 사용할 수 있는 무수한 링크를 제공하는 사이트입니다.

Planet Smalltalk - http://planet.smalltalk.org

인터넷 세계에서 발행한, 스몰토크와 관련한 블로그 항목들을 모아둔 블로그 플래닛입니다. 스몰토크와 관련한 정보를 유지할 수 있는 최고의 방법 중 하나입니다. 여러분이 GNU Smalltalk 블로그에 글을 올리는 것만으로 이 사이트에 여러분의 블로그 글을 올릴 수 있습니다.

Project Euler - http://projecteuler.net

이 사이트는 여러 사람들이 발견한 프로그래밍 문제를 게시하고, 랭킹을 올리기 위해 어떤 언어로든 문제를 해결하려는 사람들이 모이는 사이트입니다. 문제는 약간의 수학적 지식이 필요하다는 것입니다. 이 사이트는 잘 구성되어 있으며, 재미로 문제를 해결하는 곳입니다. 몇몇 문제를 해결하는 것으로 여러분의 스몰토크 지식과 알고리즘 실력을 눈에 띄게 향상시킬 수 있습니다. 또한 여러분이 공헌하여서 스몰토크 언어의 랭킹을 올려주실 수도 있습니다.

메일링 리스트

메일링 리스트는 가능한한 많은 사람들에게 여러분의 문제/제안/생각을 알릴 수 있는 방법 중 제일 좋은 방법입니다. 메일링 리스트에 메일을 보내기만 하면 모든 사람들이 여러분의 메시지를 보고, 답장을 보내줄 것입니다.

또한 http://smalltalk.gnu.org/community/ml 로 가셔서 메일링 리스트에 어떻게 가입하는지에 대해 알 수 있으며, 이전에 올린 메시지들을 검색할 수도 있습니다.

IRC

IRC(Internet Relay Chat)에서 질문을 하는 것이 GNU 스몰토크 커뮤니티로부터 가장 빨리 도움을 얻을 수 있는 방법입니다. 스몰토크와 관련한 IRC 주소와 채널은 irc.freenode.net 의 #gnu-smalltalk 입니다. 여러분이 원하는 IRC 클라이언트로 이 채널에 접속할 수 있습니다.

잠깐만... (Wait a minute...)

교황 알렉산더 4 세 (Pope Alexander VI) 그의 마지막 유언

부록 A: 프로그래밍 환경 설치

이 책을 집필할 당시 GNU 스몰토크는 주로 소스코드를 통해 배포되었습니다. 이 말인즉슨 당신이 GNU 스몰토크의 소스코드를 다운받고 컴파일해서 사용하다는 의미입니다. 당신이 운이좋다면 당신의 OS 를 위한 바이너리 배포폰을 사용할 수 있습니다. 이 장에서는 이런 모든 가능한 사항에 대해서 다뤄보겠습니다.

Linux 플랫폼에 GNU 스몰토크 설치하기

GNU 스몰토크를 당신의 Linux 시스템에서 사용하려면 크게 두가지 방법이 있습니다. 한가지는 당신의 linux 배포본에서 지원하는 패키지매니저를 사용해서 제공받는 방법이고 다른 한가지 방법은 소스코드 상태에서 GNU 스몰토크를 컴파일하는 방법입니다.

패키지 매니저를 사용한 설치

여러가지 패키지 매니저가 있지만 가장 유명한게 몇가지 있습니다. Fedora 에서는 YUM 이라고 합니다. 우분투에서는 시냅틱 패키지 매니저가 있고, Pardus 에는 PiSi 가 있습니다. 즉, 당신의 패키지 매니저에서 smalltalk 을 찾은다음 가장 높은버전의 GNU 스몰토크 패키지를 설치하면 됩니다. 버전번호는 3.0 또는 그 이상이 좋습니다만 2.x 버전만 (패키지 매니저에) 있다면 소스코드를 컴파일해서 설치하는 방법을 제안합니다. (역자주 : gentoo 의 경우는 버전업이 꽤나 잘되는 편입니다. emerge gnu-smalltalk 만 해주시면 됩니다)

소스코드를 컴파일해서 설치하기

패키지 매니저는 항상 GNU 스몰토크의 최신버전을 가지고 있지는 않습니다. 그리고 여러분의 배포판이 패키지 매니저 자체를 제공하지 않을 수도 있습니다. 그렇다면 여러분은 GNU 스몰토크를 직접 컴파일하는 방법 외에는 선택의 여지가 없습니다. 이것이 첫번째 방법보다 쉬운건 아니지만 그렇다고 어려운 방법은 아닙니다. 당신이 이 아래쪽에 있는 과정대로만 진행한다면 당신의 GNU 스몰토크 환경은 몇 분 만에 준비될 겁니다.

GNU 스몰토크는 C 와 GNU 스몰토크 프로그래밍언어로 만들어져 있습니다. 일단 C 컴파일 환경을 가지고 있어야 합니다. 대부분의 경우 이런것들은 미리 설치되어있습니다. 하지만 이렇게 (준비되어있는) 경우가 아니라면 당신의 시스템관리자에게 요청하거나 linux 배포판의 WEB 페이지등을 참고해서 (C 컴파일 환경설치법을) 학습후 설치해야 합니다. GNU 스몰토크는 쉬운 컴파일 경험을 제공하기 위해 몇 가지 GNU 도구를 사용합니다.

여기 당신의 성공적인 시도를 위한 10 가지 중요한 단계가 있습니다.

1. GNU 스몰토크는 기능을 확장하기 위한 몇 가지의 추가 패키지를 가지고 있습니다. 한 가지 예를 들자면 Blox 라는 이름의 확장패키지는 GNU 스몰토크와 함께 GUI 를 제공하기위한 방법을 제공해줍니다. 또한 Blox 는 GNU 스몰토크의 매우 유용한 도구인 클래스 브라우저를 사용하는 경우에도 필요합니다. 하지만 Blox 패키지는 Tcl 그리고 Tk 라는 두 개의 다른 소프트웨어를 필요로 합니다. 그렇기 때문에 일단 첫번째로 Tcl/TK 의 소스코드를 받아서 컴파일을 할

것입니다.

아래 사이트를 방문해주십시오.

http://www.tcl.tk/software/tcltk/download.html

그리고 Tcl 과 Tk 의 tar.gz 의 확장자를 가지는 압축된 소스코드를 다운로드 받으세요. 제 경우에는 tcl8.4.19-src.tar.gz 와 tk8.4.19-src.tar.gz 를 다운로드 받았습니다.

2. 소스코드 패키지를 작업하기 편한 폴더에 압축해제합니다.

3. 터미널을 열어서 Tcl 소스코드를 압축해제한 곳의 안쪽에 들어있는 unix 폴더로 이동합니다. 내 경우에는 이렇습니다.

```
cd /home/canol/Desktop/tcl8.4.19/unix
```

4. 아래쪽의 명령어를 차례차례 별도로 입력해주세요. 이렇게하면 GNU 스몰토크에 사용할 Tcl 의 소스코드를 컴파일할 수 있습니다. (이 단계는 몇 분 정도가 소모됩니다)

```
./configure
make
make install
```

make install 과정에서 "Permission Denied" 또는 비슷한 에러를 만난다면 super user(system 의 관라자를 말합니다)로 login 해서 명령어를 실행해야 합니다.

1. 터미널을 열어서 Tk 소스코드를 압축해제한 곳의 안쪽에 들어있는 unix 폴더로 이동합니다. 제 경우에는 이렇습니다.

```
cd /home/canol/Desktop/tk8.4.19/unix
```

2. 아래쪽의 명령어를 차례차례 별도로 입력해주세요. 이렇게하면 GNU 스몰토크에 사용할 Tk 의 소스코드를 컴파일할 수 있습니다.

```
./configure
make
make install
```

3. 이제 GNU 스몰토크를 컴파일 할 준비가 되었습니다. 경로는 이 아래쪽에 있습니다.

ftp://ftp.gnu.org/gnu/smalltalk

그리고 마지막 버전의 소스코드 패키지를 다운로드 해주세요. 제 경우 smalltalk-3.1.tar.gz 파일을 다운로드 받았습니다.

4. 소스코드 패키지를 작업하기 편한 폴더에 압축해제합니다.

5. 당신이 GNU 스몰토크 소스코드를 압축해제한 폴더로 (터미널을 열어서)이

동합니다. 제 경우에는 이렇습니다.

```
cd /home/canol/Desktop/smalltalk-3.1
```

6. 아래쪽의 명령어를 차례대로 입력해주세요. 이렇게하면 GNU 스몰토크의 소스코드를 컴파일할 수 있습니다.

```
./configure
make
make install
```

이 과정에서 아무런 에러를 발견하지 못했다면 당신은 이제 코딩할 준비가 된 것입니다.

Windows 플랫폼에 GNU 스몰토크 설치하기

이 책을 집필할 당시, GNU 스몰토크를 설치할 수 있는 Windows Installer 는 유감스럽게도 없습니다. 당신은 소스에서 컴파일을 해야합니다. GNU 스몰토크는 MingW 시스템과 함께 성공적으로 컴파일됩니다. 아래의 주소에서 찾을 수 있습니다.

http://www.mingw.org

하지만 GNU 스몰토크를 컴파일 하기 전에 의존성이 있는 별도 패키지의 다운로드가 필요할 수 있습니다. 당신은 GNU 스몰토크를 컴파일하는데 필요한 추가적인 몇 가지 정보를 "Linux 플랫폼에 GNU 스몰토크 설치하기" 에서 찾을 수 있으며 MingW 에서 컴파일을 진행하는 과정은 Linux 에서의 진행과정과 유사합니다.

하지만 Windows Installer 가 만들어지기 위한 과정이 진행되고 있습니다. Installer 는 GNU 스몰토크 공식사이트에서 얻을 수 있습니다.

http://smalltalk.gnu.org/download

만약 릴리즈된다면 말입니다. 공식 웹사이트와 메일링 리스트를 꾸준히 본다면 Windows Installer 의 진행상황을 확인하실 수 있습니다.

(역자주:결과적으로 아직 없다는 얘기입니다. Windows 에서도 컴파일을 하라는 의미입니다. 그리고 공식 사이트에서 맥의 경우는 Fink 또는 MacPorts 를 사용하라고 나와있습니다.)

부록 B: ASCII Table

Number (Decimal / Binary)	Character	Number (Decimal / Binary)	Character	Number (Decimal / Binary)	Character	Number (Decimal / Binary)	Character
0 / 000 0000	NUL (Null Character)	32 / 010 0000		64 / 100 0000	@	96 / 110 0000	`
1 / 000 0001	SOH (Start of Header)	33 / 010 0001	!	65 / 100 0001	A	97 / 110 0001	a
2 / 000 0010	STX (Start of Text)	34 / 010 0010	"	66 / 100 0010	B	98 / 110 0010	b
3 / 000 0011	ETX (End of Text)	35 / 010 0011	#	67 / 100 0011	C	99 / 110 0011	c
4 / 000 0100	EOT (End of Transmission)	36 / 010 0100	$	68 / 100 0100	D	100 / 110 0100	d
5 / 000 0101	ENQ (Enquiry)	37 / 010 0101	%	69 / 100 0101	E	101 / 110 0101	e
6 / 000 0110	ACK (Acknowledgment)	38 / 010 0110	&	70 / 100 0110	F	102 / 110 0110	f
7 / 000 0111	BEL (Bell)	39 / 010 0111	'	71 / 100 0111	G	103 / 110 0111	g
8 / 000 1000	BS (Backspace)	40 / 010 1000	(	72 / 100 1000	H	104 / 110 1000	h
9 / 000 1001	HT (Horizontal Tab)	41 / 010 1001	)	73 / 100 1001	I	105 / 110 1001	i
10 / 000 1010	LF (Line Feed)	42 / 010 1010	*	74 / 100 1010	J	106 / 110 1010	j
11 / 000 1011	VT (Vertical Tab)	43 / 010 1011	+	75 / 100 1011	K	107 / 110 1011	k
12 / 000 1100	FF (Form Feed)	44 / 010 1100	,	76 / 100 1100	L	108 / 110 1100	l
13 / 000 1101	CR (Carriage Return)	45 / 010 1101	-	77 / 100 1101	M	109 / 110 1101	m
14 / 000 1110	SO (Shift Out)	46 / 010 1110	.	78 / 100 1110	N	110 / 110 1110	n
15 / 000 1111	SI (Shift In)	47 / 010 1111	/	79 / 100 1111	O	111 / 110 1111	o
16 / 001 0000	DLE (Data Link Escape)	48 / 011 0000	0	80 / 101 0000	P	112 / 111 0000	p
17 / 001 0001	DC1 (Device Control 1)	49 / 011 0001	1	81 / 101 0001	Q	113 / 111 0001	q
18 / 001 0010	DC2 (Device Control 2)	50 / 011 0010	2	82 / 101 0010	R	114 / 111 0010	r
19 / 001 0011	DC3 (Device Control 3)	51 / 011 0011	3	83 / 101 0011	S	115 / 111 0011	s
20 / 001 0100	DC4 (Device Control 4)	52 / 011 0100	4	84 / 101 0100	T	116 / 111 0100	t
21 / 001 0101	NAK (Negative Acknowledgment)	53 / 011 0101	5	85 / 101 0101	U	117 / 111 0101	u
22 / 001 0110	SYN (Synchronous Idle)	54 / 011 0110	6	86 / 101 0110	V	118 / 111 0110	v
23 / 001 0111	ETB (End of Transmission Block)	55 / 011 0111	7	87 / 101 0111	W	119 / 111 0111	w
24 / 001 1000	CAN (Cancel)	56 / 011 1000	8	88 / 101 1000	X	120 / 111 1000	x
25 / 001 1001	EM (End of Medium)	57 / 011 1001	9	89 / 101 1001	Y	121 / 111 1001	y
26 / 001 1010	SUB (Substitute)	58 / 011 1010	:	90 / 101 1010	Z	122 / 111 1010	z
27 / 001 1011	ESC (Escape)	59 / 011 1011	;	91 / 101 1011	[	123 / 111 1011	{
28 / 001 1100	FS (File Separator)	60 / 011 1100	<	92 / 101 1100	\	124 / 111 1100	|
29 / 001 1101	GS (Group Separator)	61 / 011 1101	=	93 / 101 1101	]	125 / 111 1101	}
30 / 001 1110	RS (Record Separator)	62 / 011 1110	>	94 / 101 1110	^	126 / 111 1110	~
31 / 001 1111	US (Unit Separator)	63 / 011 1111	?	95 / 101 1111	_	127 / 111 1111	DEL (Delete)

부록 C: 연습문제 해답

1 장

1. Because of the hardware design, computers can only understand two states. We represent this two states with digits 0 and 1. Combinations of zeros and ones generate some commands that computer hardware understands. We call these command groups which provide a communication between us and computer hardware as programming language.

   We need them to communicate with computer hardware, because computer hardware can understand only two states and the programming language we use is eventually translated into the sum of combinations of this two states which is called machine language.

2. Programs written in a compiled language should be translated into machine language by other programs called compiler before the execution by the user.

   Programs written in an interpreted language is converted to machine code during the execution time by other programs called interpreter. They do not need a compilation process before they can be executed by the user but they are a little bit slower than compiled machines, though this is mostly not a problem anymore for modern computers and for not-so-performance sensitive projects.

   Although Smalltalk is designed as a language which uses a virtual machine to work, GNU Smalltalk acts more like an interpreted language.

3. A programming paradigm is how a programming language looks at the problems to be solved.

4. Yes. Such programming languages are called multi-paradigm programming languages.

   No, GNU Smalltalk allows programmers to use only object-oriented programming paradigm.

5. **From decimal to binary:**

$$543/2 = 271 \quad (remainder\ \mathbf{1})$$

$$271/2 = 135 \quad (remainder\ \mathbf{1})$$

$$135/2 = 67 \quad (remainder\ \mathbf{1})$$

$$67/2 = 33 \quad (remainder\ \mathbf{1})$$

$$33/2 = 16 \quad (remainder\ \mathbf{1})$$

$$16/2 = 8 \quad (remainder\ \mathbf{0})$$

$$8/2 = 4 \quad (remainder\ \mathbf{0})$$

$$4/2=2 \quad (remainder\,\mathbf{0})$$

$$2/2=\mathbf{1} \quad (remainder\,\mathbf{0})$$

So the answer is: **1000011111**

From decimal to octal:

We can use the same method to convert it from base-10 to base-8 but we learned an easier way, so let's use it:

$$(543)_{10}=\underbrace{001}_{1}\,\underbrace{000}_{0}\,\underbrace{011}_{3}\,\underbrace{111}_{7}=(1037)_8$$

From decimal to hexadecimal:

$$(543)_{10}=\underbrace{0010}_{2}\,\underbrace{0001}_{1}\,\underbrace{1111}_{F}=(21F)_{16}$$

6. **From binary to decimal:**

$$(10110100)_2=0\cdot 2^0+0\cdot 2^1+1\cdot 2^2+0\cdot 2^3+1\cdot 2^4+1\cdot 2^5+0\cdot 2^6+1\cdot 2^7$$
$$(10110100)_2=0+0+4+0+16+32+0+128=(180)_{10}$$

From binary to octal:

$$(010110100)_2=\underbrace{010}_{4}\,\underbrace{110}_{6}\,\underbrace{100}_{4}=(464)_8$$

From binary to hexadecimal:

$$(10110100)_2=\underbrace{1011}_{B}\,\underbrace{0100}_{4}=(B4)_{16}$$

7. **From hexadecimal to binary:**

$$(A93F)_{16}=\underbrace{1010}_{A}\,\underbrace{1001}_{9}\,\underbrace{0011}_{3}\,\underbrace{1111}_{F}=(1010100100111111)_2$$

From hexadecimal to octal:

We can look at place value of each digit, treat them as octal numbers and sum them up again in octal rules but this is hard for humans because we used to and tend to treat numbers as they are decimal so converting a hexadecimal number to binary and converting it to octal is usually simpler:

$$(A93F)_{16}=\underbrace{001}_{1}\,\underbrace{010}_{2}\,\underbrace{100}_{4}\,\underbrace{100}_{4}\,\underbrace{111}_{7}\,\underbrace{111}_{7}=(124477)_8$$

From hexadecimal to decimal:

Now, we can use the method we mentioned above:

$$(A93F)_{16} = 15 \cdot 16^0 + 3 \cdot 16^1 + 9 \cdot 16^2 + 10 \cdot 16^3$$
$$(A93F)_{16} = 15 + 48 + 2304 + 40960 = (43327)_{10}$$

8. Binary files are composed of zeros or ones placed according to a format specification. They can specify any kind of information from a document to image or video. Text files are however specially designed binary files to contain only character data using a character set and encoding.

   .zip and .avi files may be given as examples of other binary files.

   You can try to open a file with a text editor and if it contains only meaningful characters and not funny symbols all around, it is most probably a text file. Also some text editors might detect that a file is not written in a recognized text encoding and warn the user about it.

9. Text editors are designed only to work with text files while word processors have most probably have their own binary file format to save documents. Like .doc for Microsoft Word and .odt for OpenOffice.org Writer. Word processors have a lot of fancy formatting tools to write a text in bold, italic, colored etc. which we don't use when writing text files.

   Most word processors have options to work with text files but they have so much unnecessary features we won't use that using them is like using a limousine to harvest.

2 장

1.

```
"answer_2_1.st"

Transcript show: '  /\  '; cr.
Transcript show: ' /  \ '; cr.
Transcript show: '/    \'; cr.
Transcript show: '\    /'; cr.
Transcript show: ' \  / '; cr.
Transcript show: '  \/  '; cr.
```

```
  /\
 /  \
/    \
\    /
 \  /
  \/
```

2. When we have a small task to complete or when we are experimenting things, using terminal to enter the program instructions is very practical. But if we have to write a big program, saving it to hard disk by writing its source code into a text file will make it easy for us to work on it later on. Also saving the source code provides us the opportunity to use that program again whenever we want.

3. Comments provide the reader of the source code additional information so that he/she understands the code faster. The reader might be another programmer or the programmer who wrote that program himself. Even we might not remember what we intended to do with a specific part of the source code when reading it a few weeks later. Also in GNU Smalltalk they are used for documentation purposes which we will see later.

   In GNU Smalltalk whatever we write between double quotes are treated as comments by interpreter, except double quotes itself. If we want to write double quotes in a comment, we should write two double quotes, consecutively or we might use single quotes if appropriate for reader.

4. We can do that with our current knowledge by writing every single number manually like:

```
1 printNl.
2 printNl.
3 printNl.
.
.
.
998 printNl.
999 printNl
1000 printNl.
```

   But of course, this is not practical and nobody can force us to do that. Later on this book, we will learn about *loops* and see how we can achieve this task easily in just one line of code.

   If we imagine about it, we will probably come to a point that: "We should be able to define a *range* and make the `printNl` command act on that range.", which is the exact solution we will see.

5. Everything the computer generously give us is an output. It can be an image, a sound, a movement etc. So, actually, beside our monitor image which is in the form of a terminal text in this case, our speakers or a printer may also be classified as output devices.

   Also, keyboard is not the only input device. Every device which give us the opportunity to send information to the computer is an input device. For example, a mouse, a scanner, a web cam, a touch screen may all be specified as input devices.

   Programming languages usually give us to change standard input and output devices so that the computer and the user interacts with each other using different ways and GNU Smalltalk is not an exception.

3 장

1. An object is a unit which represents a *thing* in the program. It has some definitions

how to respond to certain messages it receives from outside (usually from us or from other objects) which are called methods. We communicate with objects via messages we send to invoke the methods inside them. Classes are the templates of the objects we want to create. We first define a class, then produce objects from it.

2. Giving a name to an object is called assignment and we call this names variables in programming jargon. This is because although a name refers to only one object, the object it refers to can be changed any time.

   We need them to hold a data for later use in a program. The data might expected to be changing in different parts of a program so giving it a name would be easier for us instead of dealing with its value, directly.

3. Messages are classified as unary, binary and keyword messages. We can count `reverse` and `asUppercase` as examples of unary messages; count `+` and `*` as binary messages; count `at:put:` and `removeKey:` as keyword messages.

4. We can deal with natural numbers like `3`, `10`, `25`; real numbers like `3.5`, `8.2`, `0.4e5` or fractions like `4/7`, `8/13`, `25/138`.

5.

```
"answer_3_5.st"

| theNumber |

Transcript show: 'Please enter a number to get its cube: '.
theNumber := stdin nextLine.

Transcript show: 'The cube of ', theNumber, ' is ',
(theNumber asInteger squared * theNumber asInteger)
printString, '.'; cr.
```

```
Please enter a number to get its cube: 3
The cube of 3 is 27.
```

6.

```
"answer_3_6.st"

| theNumbers arithmeticAverage |

Transcript show: 'Please enter two numbers separated by
space to get their arithmetic average: '.
theNumbers := stdin nextLine tokenize: ' '.

arithmeticAverage := ((theNumbers at: 1) asInteger +
(theNumbers at: 2) asInteger) / 2.
```

```
Transcript show: 'The arithmetic average of ', (theNumbers
at: 1), ' and ',  (theNumbers at: 2), ' is ',
arithmeticAverage printString, '.'; cr.
```

```
Please enter two numbers separated by space to get their
arithmetic average: 10 20
The arithmetic average of 10 and 20 is 15.
```

7.

```
"answer_3_7.st"

| definitions theWord |

definitions := Dictionary new.

definitions at: 'programming language' put: 'We call the
command groups which provide a communication between us and
computer hardware as programming languages.'.
definitions at: 'source code' put: 'The code we wrote in a
programming language (not the result of a compilation or
interpretation process) is called source code.'.
definitions at: 'virtual machine' put: 'The converting
process from byte-code to machine code is done by programs
called virtual machines.'.
definitions at: 'cross-platform' put: 'Cross-platform is the
name for being able to run a software on different computer
architectures, like different operating systems or different
processors.'.

Transcript show: 'Please enter a word to get its definition:
'.

theWord := stdin nextLine.

Transcript show: theWord, ': ', (definitions at: theWord
asLowercase); cr.
```

```
Please enter a word to get its definition: programming
language
programming language: We call the command groups which
provide a communication between us and computer hardware as
programming languages.
```

We put an `asLowercase` message to the last expression because the keys are case sensitive and via this method, the program will be able to find the correct key even if the user enters, for example, *Programming Language*.

8. In `Arrays`, we have a collection of data which is held in order and accessed by a numbered index. In `Dictionarys`, we also have a collection data but not held in any

order. So, one of the reasons might be about the structure we want to use. For example, we can create a `Students` object as `Array` and reach the information of a student by a numbered index like `Student at: 1`. Or we can choose to keep students according to their name via a `Dictionary` class and reach a student's information by writing `Student at: Canol Gökel`.

An other reason might about the performance. `Array`s keep their data consecutively inside the computer memory and the access time for an individual element is always the same while in `Dictionary`s the time increases as the element size increases. Though it is not a significant increase for not too big `Dictionary`s.

4 장

1. The ultimate goal of control messages is to give the user opportunity to change the flow of the program instructions. Otherwise we won't be able to make decisions or execute a part of the program more than once.

   There are two types of control messages: selective and repetitive. `ifTrue:` and `ifFalse:` are examples of selective control messages while `whileTrue:` and `to:do:` are examples of repetitive control messages.

2. Blocks are groups of expressions which we can pass to control messages. They are one of the things that make it possible to have control messages.

3. Definite loops are loops we know how many times it will repeat itself. The end of indefinite loops, however, depends on a certain condition to occur. `to:do:` is an example of definite loop messages while `whileTrue:` is an example of indefinite loops.

4.

```
"answer_4_4.st"

| theLength |

Transcript show: 'Please enter the length of the edge of the
diamond: '. t
heLength := stdin nextLine asInteger.

1 to: theLength do: [ :x |
    1 to: (theLength - x) do: [ :y |
        Transcript show: ' '.
    ].

    Transcript show: '/'.

    1 to: ((x - 1) * 2) do: [ :y |
        Transcript show: ' '.
```

```
    ].

    Transcript show: '\'; cr.
].

1 to: theLength do: [ :x |
    1 to: (x - 1) do: [ :y |
        Transcript show: ' '.
    ].

    Transcript show: '\'.

    1 to: ((theLength - x) * 2) do: [ :y |
        Transcript show: ' '.
    ].

    Transcript show: '/'; cr.
].
```

```
Please enter the length of the edge of the diamond: 5
    /\
   /  \
  /    \
 /      \
/        \
\        /
 \      /
  \    /
   \  /
    \/
```

5.

```
"answer_4_5.st"

| theNumber isPrime i |

Transcript show: 'Please enter a number: '.
theNumber := stdin nextLine asInteger.

i := 2.

[theNumber \\ i = 0] whileFalse: [
    i := i + 1.
].

(i = theNumber) ifTrue: [
    Transcript show: theNumber printString, ' is a prime
number.'; cr.
```

```
] ifFalse: [
    Transcript show: theNumber printString, ' is not a prime
number. It is devidable by ', i printString, '.'; cr.
].
```

```
Please enter a number: 3
3 is a prime number.
```

6.

```
"answer_4_6.st"

99 to: 1 by: -1 do: [ :x |
    Transcript show: x printString, ' bottles of beer on the
wall, ', x printString, ' bottles of beer.'; cr.

    (x > 1) ifTrue: [
        Transcript show: 'Take one down and pass it around,
', (x - 1) printString, ' bottles of beer on the wall.'; cr.

    ] ifFalse: [
        Transcript show: 'Take one down and pass it around,
no more bottles of beer on the wall.'; cr.
    ].

    Transcript cr.
].

Transcript show: 'No more bottles of beer on the wall, no
more bottles of beer.'; cr.
Transcript show: 'Go to the store and buy some more, 99
bottles of beer on the wall.'; cr.
```

7. With definite loop:

```
"answer_4_7_a.st"

1 to: 10 do: [:x | x printNl]
```

```
1
...
10
```

With indefinite loop:

```
"answer_4_7_b.st"

| x |

x := 1.
```

```
[x <= 10] whileTrue: [
    x printNl.
    x := x + 1.
]
```

```
1
...
10
```

First one feels the right one because we didn't have to declare a variable and didn't have to increment it manually. Also we didn't spend time thinking about the right conditional, we just wrote the range. This shows we should write a loop using definite loop techniques whenever the range of the loop is known before we enter into it.

8. It is impossible to write this program with definite loops because we don't know when the user is going to finish the program.[5] In other words, how many times the loop will repeat executing itself depends on the task user want to accomplish, which the program cannot know, so we (as the programmer) don't know the range of the loop before entering into it.

   By using indefinite loop:

```
"answer_4_8.st"

| newNumber howManyNumbers sum arithmeticAverage |

newNumber := 0.

howManyNumbers := 0.
sum := 0.
arithmeticAverage := 0.

[newNumber = 'finish'] whileFalse: [
    Transcript show: 'Arithmetic averages of the numbers so
far is: ', arithmeticAverage printString; cr.

    Transcript show: 'Please enter a new number or "finish"
to exit '.
    newNumber := stdin nextLine.

    (newNumber isNumeric) ifTrue: [
        sum := sum + newNumber asInteger.
        howManyNumbers := howManyNumbers + 1.
        arithmeticAverage := sum / howManyNumbers.
    ].
```

5 Actually, it is possible but it is hard and unnecessary.

```
    Transcript cr.
].
```

```
Arithmetic averages of the numbers so far is: 0
Please enter a new number or "finish" to exit: 2

Arithmetic averages of the numbers so far is: 2
Please enter a new number or "finish" to exit: 4

Arithmetic averages of the numbers so far is: 3
Please enter a new number or "finish" to exit: finish
```

9.

```
1 to: 1000 do: [:x | x printNl].
```

```
1
...
1000
1
```

5 장

1. Polymorphism is the name of the concept to determine which method to execute according to the type of the class of the object we sent the message.

   Observing the attributes and behavior of the ancestor class when deriving a new class from it is called inheritance.

   Encapsulation is hiding the inner working details of an object from outside world.

2.

```
"answer_5_2.st"

Object subclass: Human [
    | name age |

    setName: aName [
        name := aName.
    ]

    getName [
        ^name
    ]

    setAge: anAge [
        age := anAge.
    ]

    getAge [
```

```
        ^age
    ]

    introduceYourself [
        Transcript show: 'Hello, my name is ', name, ' and
I''m ', age printString, ' years old.'; cr.
    ]

    > aHuman [
        ^age > aHuman getAge
    ]

    < aHuman [
        ^age < aHuman getAge
    ]

    = aHuman [
        ^age = aHuman getAge
    ]
]

Human subclass: Man [
    | money handsomeness |

    setMoney: amountOfMoney [
        "Amount of money out of 10"

        money := amountOfMoney.
    ]

    getMoney [
        ^money
    ]

    setHandsomeness: rateOfHandsomeness [
        "Handsomeness rate out of 10"

        handsomeness := rateOfHandsomeness.
    ]

    getHandsomeness [
        ^handsomeness
    ]

    > aMan [
        (self getName = 'Canol Gökel') ifTrue: [
            ^true
```

```
        ] ifFalse: [
            ^(self getMoney + self getHandsomeness) > (aMan
getMoney + aMan getHandsomeness)
        ]
    ]

    < aMan [
        (self getName = 'Canol Gökel') ifTrue: [
            ^false
        ] ifFalse: [
            ^(self getMoney + self getHandsomeness) < (aMan
getMoney + aMan getHandsomeness)
        ]
    ]
    = aMan [
        (self getName = 'Canol Gökel') ifTrue: [
            ^false
        ] ifFalse: [
            ^(self getMoney + self getHandsomeness) = (aMan
getMoney + aMan getHandsomeness)
        ]
    ]
]

Human subclass: Woman [
    | honesty generosity |

    setHonesty: rateOfHonesty [
        "Honesty rate out of 10"

        honesty := rateOfHonesty.
    ]

    getHonesty [
        ^honesty
    ]

    setGenerosity: rateOfGenerosity [
        "Generosity rate out of 10"

        generosity := rateOfGenerosity.
    ]

    getGenerosity [
        ^generosity
    ]
```

```
    > aWoman [
        ^(self getHonesty + self getGenerosity) > (aWoman
getHonesty + aWoman getGenerosity)
    ]

    < aWoman [
        ^(self getHonesty + self getGenerosity) < (aWoman
getHonesty + aWoman getGenerosity)
    ]

    = aWoman [
        ^(self getHonesty + self getGenerosity) = (aWoman
getHonesty + aWoman getGenerosity)
    ]
]

| man1 man2 man3 woman1 woman2 |

man1 := Man new.
man1 setName: 'Michael Cooper'.
man1 setAge: 32.
man1 setMoney: 9.
man1 setHandsomeness: 7.

man2 := Man new.
man2 setName: 'Paul Anderson'.
man2 setAge: 28.
man2 setMoney: 7.
man2 setHandsomeness: 8.

man3 := Man new.
man3 setName: 'Canol Gökel'.
man3 setAge: 24.
man3 setMoney: 1.
man3 setHandsomeness: 1.

woman1 := Woman new.
woman1 setName: 'Louise Stephney'.
woman1 setAge: 26.
woman1 setHonesty: 6.
woman1 setGenerosity: 7.

woman2 := Woman new.
woman2 setName: 'Maria Brooks'.
woman2 setAge: 32.
woman2 setHonesty: 8.
woman2 setGenerosity: 6.
```

```
(man1 > man2) printNl.
(man1 < man2) printNl.
(man3 > man1) printNl.
(woman1 > woman2) printNl.
(woman1 < woman2) printNl.
```

```
true
false
true
false
true
```

3.

```
"answer_5_3.st"

Number extend [
    cubed [
        ^self * self * self
    ]
]

3 cubed printNl.
```

```
27
```

4. These two keywords are used to refer to the receiver object of the message, inside the class definition. Whenever we send a message to `self` or `super` keyword, the interpreter sends the message to the current object in context.

   The difference of `super` keyword from `self` is that when we send a message to `super`, the search of the method begins from the superclass of the receiver object. So the method inside the superclass is invoked instead of the method of the receiver object's class.

인덱스

제 7 장 후문

이제 GNU 스몰토크를 이용하여 컴퓨터 프로그래밍의 기본을 배우는 우리의 여정은 끝났습니다. 여러분이 이 책을 즐겼길 바랍니다. 이 책은 저자정보 절에서 찾을 수 있는 주소의 e 메일로 항상 피드백을 받습니다.

저자 정보

Canol Gökel 은 터키의 Hacettepe University 전기전자공학부 학생이며, Hacettepe Robotics 학회의 구성원입니다. 컴퓨터와 마이크로컨트롤러 프로그래밍을 좋아하며, 새로운 프로그래밍 언어 공부를 하는 것을 좋아합니다. 다음 이메일 주소로 이메일을 보내주세요. canol@canol.info, 비둘기 서신으론 다음 주소로 보내주세요. Hacettepe Universitesi Beytepe Kampusu, Ogrenci Evleri, J Blok, No: Z-09, Ankara/Turkey