예제로 배우는 스퀵

Andrew P. Black Stéphane Ducasse

Oscar Nierstrasz Damien Pollet

with Damien Cassou and Marcus Denker

Version of 2009-09-29

이 책은 스위스 베른대학 컴퓨터 공학 및 응용 수학 연구소에서 호스팅하는 SqueakByExample.org 에서 무료로 다운로드 하실 수 있습니다.

출 판 : Square Bracket Associates, Switzerland. SquareBracketAssociates.org

ISBN 978-3-9523341-0-2

초 판 : 2007년 09월.

수정판 : 2008년 03월, 2008년 05월, 2009년 05월.

2011년 04월 19일 삽화 정리.

차 례

그림 차례

표 차례

Listings

들어가는 글

스퀵(Squeak)이란?

스퀵(Squeak)은 스몰토크(Smalltalk) 프로그래밍 언어이며 완전한 기능을 갖춘 최신 오픈소스 환경입니다. 스퀵은 굉장히 이식성이 높기 때문에 디버그, 분석 그리고 변경이 용이한 스몰토크언어로 스퀵전체의 가상머신을 작성했습니다. 스퀵은, 멀티미디어 응용 프로그램과 교육 플렛폼에서 시작해서, 상업적인 웹 개발 환경에 이르기까지 광범위한 영역에 걸쳐 혁신적인 프로젝트를 진행하기 위한 훌륭한 도구입니다.

누가 이 책을 읽어야 할까요?

이 책은 기초부터 시작해서 고급 주제까지 진행하면서, 스퀵의 다양한 모습들을 보여줍니다.

이 책은 프로그램 작성법을 가르쳐 주는것은 아닙니다. 독자 여러분은 프로그래밍 언어에 어느 정도 익숙해야 합니다. 객체지향 프로그래밍에 대한 약간의 배경지식을 가지고 계신다면 도움이 됩니다.

이 책은 스퀵프로그래밍 환경과, 언어*programming language*, 그리고 관련 도구를 소개할 것입니다. 이책을 통해 일반적인 용어들과 실제사례등을 만나겠지만, 객체지향 설계가 아닌 기술에 중점을 두었습니다. 우리는 가능한 모든 지면을 통해 수많은 예를 보여드리겠습니다. (Alec Sharp의 놀라운 스몰토크책[1]

에서 영향을 받았습니다.)

웹에서 무료로 볼 수 있는 수많은 스몰토크책이 있지만, 그들 중 어떤 책도 스퀵에 대해 구체적으로 다루지 않았습니다.

예제를 보시려면 다음을 방문해 주십시오:
stephane.ducasse.free.fr/FreeBooks.html

조언 한마디

당신이 스몰토크의 일부를 바로 이해하지 못한다고 해도, 포기하지 마세요. 스몰토크의 모든 것을 알아야 할 필요는 없습니다! Alan Kingt 는 다음과 같은 원리를 표현하였습니다.[1]

> **너무 신경쓰지 마세요.** 스몰토크프로그래머는 새로 배우는 단계에서 종종 어려운 상황을 만납니다. 스몰토크를 사용하기 전에 이 프로그램의 동작 원리에 대한 모든 세부사항들을 이해할 필요가 있다고 생각하기 때문이죠. 이것은 스몰토크에서 `Transcript show: 'Hello world'` 를 완벽히 이해하기 전에 꽤 긴 시간이 걸린다는 의미가 되기도 합니다. 객체지향을 통한 큰 발전 요소중 하나는 "이것이 어떤 식으로 동작하지?"에 대한 질문에 "나는 상관 안 해"라는 대답이 가능하다는 것입니다.

공개 서적

이 책은 다음의 내용을 통해 공개한 책입니다:

- 이 책의 컨텐츠는 크리에이티브 커먼즈 저작자표시-동일변경조건허락 (by-sa) 라이선스 하에 출판됩니다. 간단히 말해, 사용자는 다음 URL 에 실린 조건들을 준수한다면, 이 책을 자유롭게 공유하고 사용할 수 있습니다:
  creativecommons.org/licenses/by-sa/3.0
- 이 책은 스퀵의 핵심만을 설명합니다. 우리는 이상적인 방향을 위해, 우리가 설명하지 않은 스퀵의 내용들을 당신이 기고해주기를 권장하고 싶습니다. 당신이 이 수고에 참여해주시기를 원하신다면, 우리에게 연락해 주십시오. 우리는 이 책이 성장하는 것을 보고 싶습니다!

[1] www.surfscranton.com/architecture/knightsPrinciple.htm

더 많은 내용을 보시려면, 스위스 Bern 대학교의 컴퓨터 과학 및 응용 수학 연구소가 제공하는 이 책의 웹사이트 SqueakByExample.org 를 방문해 주십시오.

스퀵커뮤니티

스퀵커뮤니티는 친근하며 활동적입니다. 독자가 찾고싶어할만한 유용한 자료에 대해 간단한 목록을 적어두었습니다.

- www.squeak.org 는 스퀵의 메인 웹사이트 입니다. (스퀵의 최고봉인 eToy 환경의 내용에 대해 제공하지만, 방문 대상이 초등학교 교사들인 www.squeakland.org와 혼동하지 마십시오)
- www.squeaksource.com 은 스퀵프로젝트를 위한 사이트이며, 소스포지 (SourceForge) 와 동일합니다.
- wiki.squeak.org/squeak 은 스퀵에 대한 최신의 정보를 갖고 있는 위키입니다.

메일링 리스트 정보. 수많은 메일링 리스트가 있는데, 때로는 약간 지나치게 활동적일 수도 있습니다. 메일을 메일 계정 용량에 넘치도록 받기를 원하는 건 아니겠지만, 그래도 메일링 리스트에 등록되어 있기를 원하신다면, 메일링 리스트를 검색하기 위해 news.gmane.org 또는 www.nabble.com/squeak-f14152.html을 사용해 보십시오.

squeakfoundation.org/mailman/listinfo squeakfoundation.org/mailman/listinfo 목록에서 전체 스퀵메일링 리스트를 찾을 수 있습니다.

- 스퀵-*dev*는 다음의 주소에서 찾을 수 있는 개발자의 메일링 리스트를 참조함을 참고하십시오 :
  news.gmane.org/gmane.comp.lang.smalltalk.squeak.general

- *Newbies(초보자)* 여러분은 초보자들이 어떤 질문이라도 할 수 있는 친근한 메일링 리스트를 참조하십시오: news.gmane.org/gmane.comp.lang.smalltalk.squeak.beginners (우리는 모두 스퀵의 일부 측면에 있어 모두 초보자이기에, 배워야 할 내용이 너무나 많습니다!)

IRC. 빠른 답이 필요한 질문이 있습니까? 전 세계에 있는 스퀵사용자를 만나고 싶으십니까? Irc.freenode.net에 있는 "#squeak" 채널은 IRC 채널에서 긴 시간의 회의에 참여할 수 있는 훌륭한 장소입니다. 잠시 들르셔서 "Hi!"라고 인사말을 남겨주세요!

다른 사이트. 오늘날 다양한 방식으로 스퀵커뮤니티를 지원하는 여러 곳의 웹사이트가 있습니다. 그 사이트 중 몇 군데를 적어두었습니다.

- people.squeakfoundation.org는 스퀴커를 위한 "advogato.org"와 같은 종류의 SqueakPeople 사이트입니다. 이 웹사이트는 게시글, 다이어리 그리고 흥미로운 신용 통계 시스템을 제공합니다.
- planet.squeak.org 는 RSS 수집 역할을 하는 PlanetSqueak 사이트입니다. 이 사이트는 스퀵에 대한 수많은 자료들을 얻을 수 있는 좋은 곳입니다. 스퀵개발자들과 스퀵에 관심있는 다른 사람들이 기고하는 최신 블로그 내용이 있습니다.
- www.frappr.com/squeak 은 전 세계에 있는 스퀵사용자를 추적하는 사이트입니다.

예시와 연습

이 책에서 두 가지 특별한 규칙을 사용합니다.

가능한 많은 예시를 제공하려고 했습니다. 특히, 실행 가능한 코드 일부를 보여주는 많은 예시를 넣었습니다. 우리는 사용자가 프로그램식(expression)

과 `print it`을 선택할 때 얻을 수 있는 결과를 나타내기 위해 `-->` 심볼을 사용하였습니다:

```
3 + 4  ⟶  7
```

이러한 코드 일부를 가지고 스퀵을 즐기시려는 경우, 이 책의 웹사이트에서 모든 예제 코드와 텍스트 파일을 다운로드 할 수 있습니다:SqueakByExample.org

우리가 사용하는 두 번째 규칙은 사용자가 무엇인가를 해야 할 경우  아이콘을 표시하는 것입니다.

 다음 장으로 가서 내용을 읽어보십시오!

감사의 말

스몰토크에 대한 칼럼 일부 번역을 허락해 준 Hilaire Fernandes와 Serge Stinckwich, 스트림에 대한 장을 기고해준 Damien Cassou에게 감사드리고 싶습니다. 또한 스퀵로고를 사용하게 해준 Tim Rowledge와 원본 표지 그림을 사용하게 해준 Frederica Nierstrasz에게도 감사드립니다.

특히 첫 번째 출판 초안에 조언을 해준 Renggli와 Orla Greevy에게 감사를 표현하고 싶습니다.

이 책의 웹사이트 제공을 위해 이 오픈소스 프로젝트를 자비롭게 지원해준 스위스 bern 대학교 관계자 여러분께 감사를 드립니다.

또한 이 프로젝트에 대해 열정적인 지원을 해주고 이 책의 첫 판에서 발견한 오류를 알려준 스퀵커뮤니티 여러분께 감사를 드립니다. 마지막으로, 우리가 사용할 수 있도록 놀라운 개발 환경을 만들어준 가장 먼저 스퀵을 개발한 팀께 감사드립니다.

제 I 편

시작하기

제 1 장

스퀵으로의 짧은 여행

이 장에서는 여러분이 스퀵환경에서 편안함을 느끼실 수 있도록 도와드리기 위해 스퀵으로의 고품격의 여행을 보내드릴 것입니다. 스퀵을 편안하게 다루어 보실 수 있는 많은 기회들이 있으므로, 이 장을 읽으시는 동안 가까운 곳에 컴퓨터를 준비하는 것이 좋습니다.

여러분이 스퀵에서 무엇인가를 해야 할 때, 본문에서 이 아이콘  을 사용하겠습니다. 특히, 여러분은 시스템과 상호작용하는 다양한 방법을 배우기 위해 스퀵을 실행할 것입니다. 또한 새로운 메서드를 정의하고 오브젝트를 만들며 여기에 메시지를 보내는 법을 배우겠습니다.

1.1 시작하기

스퀵은 www.squeak.org에서 무료로 다운로드 하실 수 있습니다. 4개의 파일로 구성된(그림 1.1 참조) 3개의 파트를 다운로드해야 합니다.

1. 가상 머신*(VM)*은 스퀵시스템의 일부분이며 각각의 운영체제와 프로세서에 따라 차이가 있습니다. 미리 컴파일한 가상 머신은 모든 주요 컴퓨터 환경에서 사용할 수 있습니다. 그림 1.1 에서, *Squeak 3.8.15beta-1U.app* 이라고 하는 Mac 용 VM을 볼 수 있습니다.

그림 1.1: 스퀵다운로드 파일.

2. *source* 파일은 스퀵에서 자주 변경되지 않는 부분을 포함한 모든 부분에 대한 소스 코드(source code)를 포함하고 있습니다. 그림 1.1 에서는, 이것을 *SqueakV39.sources*라고 부르고 있습니다. 참고로 SqueakV39.sources 파일은 스퀵3.9 이상의 버전에서만 사용할 수 있습니다. 이전 버전에서는, 예를 들어, 스퀵3.0에서 3.8 버전까지 *SqueakV3.sources* 파일처럼 주 버전에 맞는 sources 파일을 사용하십시오.

3. 현재 *system image*는 실행중인 스퀵시스템의 정지화면 스냅샷입니다. 이미지는 시스템의 모든 객체(객체인 클래스와 메서드를 포함) 상태를 포함하고 있는 *.image* 파일과, 시스템의 소스 코드에 대한 모든 변경 사항 로그를 지닌 *.change* 파일, 두 가지 파일로 구성되어 있습니다. 그림 1.1 에서 *Squeak3.0-final-7067* image 파일과 change 파일을 확인할 수 있습니다. 실제로 이 책에서는 약간 다른 image 파일을 사용할 것입니다.

 사용자의 컴퓨터에 스퀵을 다운로드 하고 설치하십시오. 예제로 배우는 스퀵웹페이지에서 제공하는 *image* 파일 사용을 권합니다.[1]

이 책에 있는 대부분의 소개 자료는 어떤 버전에서든 작동하므로, 스퀵을 설치하셨다면, 계속해서 사용하실 수 있습니다. 하지만, 사용자의 시스템에 나타난 스퀵의 모양새, 동작 방식이 이 책에 설명한 스퀵의 모양새, 동작 방식과의 차이점이 있더라도 놀라지는 말아주세요. 한편, 당신이 지금 막 스퀵을 처음 다운로드 하신다면, 이 책에서 제공된 *Squeak by Example* image 파일을 사용하시는 것이 더 좋은 방법이 될겁니다.

[1] SqueakByExample.org 를 방문하고 가장자리에서 "Download Squeak"을 찾으십시오.

스퀵으로 작업할 때, image 파일과 changes 파일이 수정되므로, 쓰기 가능한 파일인지 확인하셔야 합니다. 항상 이 두 가지 파일들이 함께 있도록 해주셔야 합니다. 스퀵은 사용자가 작업한 객체들을 저장하고, 소스코드를 바꾼 내용을 로그로 남기므로, 절대로 텍스트 편집기를 통해 이들 두 파일을 직접 편집하지 마십시오. 새로운 이미지에서 시작하고 당신의 코드를 다시 불러올 수 있도록, 다운로드한 image 파일과 change 파일의 사본을 항상 만들어 놓는 것이 바람직합니다.

sources 파일과 가상 머신은 읽기 전용으로 만들어서 다른 사용자와 공유할 수 있습니다. 이 모든 파일은 같은 디렉터리에 있어야 하지만, 모든 사용자가 접근할 수 있도록 가상 머신과 sources 파일을 개별 디렉터리에 넣을 수도 있습니다. 어떤 방식으로든 여러분의 작업 방식과 운영 체제에 가장 잘 맞는 방식으로 수행해 주십시오.

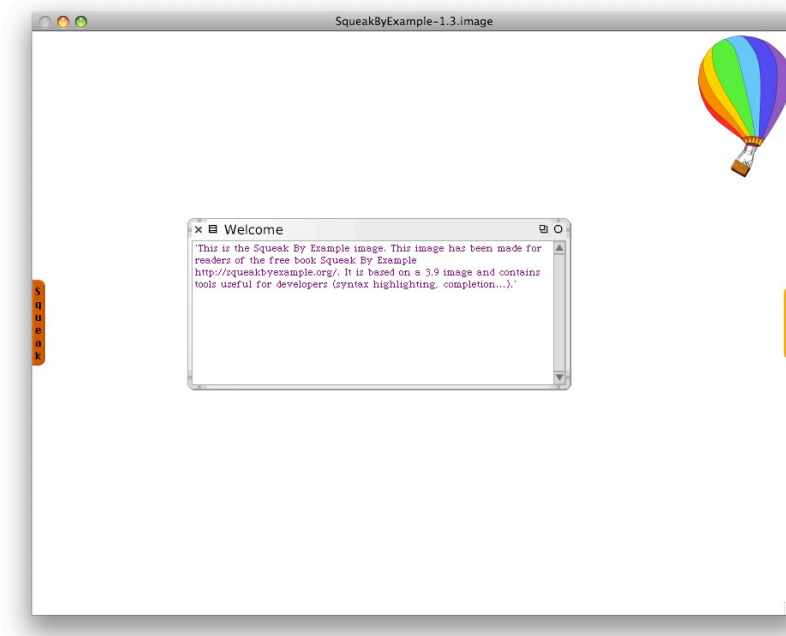


그림 1.2: 초기 SqueakByExample.org image 파일

스퀵실행하기. 스퀵을 시작하려면, 어떤 방식으로든 사용자의 운영체제에 따른 방식으로 실행하셔야 합니다. *.image* 파일을 가상 머신의 아이콘 위에 끌어다 놓으시거나, *.image* 파일을 더블 클릭 하시거나, 명령줄에서 경로가 따라오고 *.image*로 끝나는 가상 머신의 이름을 입력해 주십시오. (사용자의 시스템에 여러 가지 스퀵가상 머신을 설치했다면, 운영 체제가 자동으로 알맞은 것을 고르지 못 할 때가 있습니다. 이런 경우, 가상 머신에 이미지를 끌어다 놓거나 명령줄을 사용하는 것이 더 안전합니다.)

스퀵이 일단 실행중이라면, 가능한 경우 몇 개정도 열려있는 Workspace 창 (그림 1.2 참조)과 한 개의 큰 창이 보여야만 하며, 이것이 어떻게 진행될 지는 확실하지 않습니다! 사용자는 메뉴 표시줄이 없거나 또는 적어도 한 개의 쓸만한 메뉴 표시줄 조차도 없다는 사실을 눈치채실것입니다. 대신 스퀵은 상황에따른 팝업메뉴를 자주 사용하게 합니다.

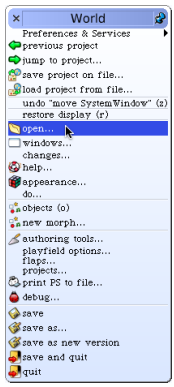
스퀵을 시작하십시오. *Workspace* 창의 왼쪽 모서리 상단에 *"x"* 를 클릭하여 열려있는 *Workspace* 를 닫을 수 있습니다. 오른쪽 모서리 상단의 *"O"* 를 클릭하면 (나중에 다시 확장할 수 있도록) 창들을 최소화 할 수 있습니다.

첫 번째 상호작용. 시작하기에 좋은 위치는 그림 1.3a 에 보이는 월드 메뉴입니다.

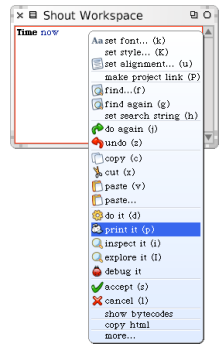
월드 메뉴를 보이기 위해 메인 창의 배경 위에서 마우스를 클릭하고, 새로운 *Workspace* 를 만들기 위해 `open… ▷ workspace` 을 선택하십시오.

스퀵은 본래 3-버튼 마우스 를 가진 컴퓨터를 대상으로 설계했습니다. 여러분의 마우스의 버튼 갯수가 3보다 적다면, 추가 버튼을 시뮬레이션 하기 위해 마우스를 클릭하는 동안 추가 키를 눌러야 합니다. 두 개의 버튼을 가진 마우스는 스퀵과 꽤 잘 맞지만, 단지 하나의 버튼을 가진 마우스만을 갖고 있다면, 클릭이 가능한 스크롤 휠과 두 개의 마우스 버튼을 가진 마우스의 구입을 심각하게 고려해 보셔야 합니다. 이렇게 하면 스퀵을 훨씬 더 쾌적하게 사용할 수 있도록 도움됩니다.

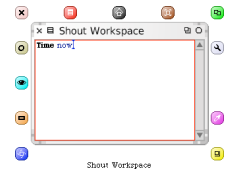
그림 1.4: 글쓴이의 마우스입니다. 스크롤 휠을 클릭하면 파랑 버튼을 활성화합니다.



(a) 월드메뉴 (The world menu)



(b) 상황 메뉴 (The contextual menu)



(c) 모픽 할로 (Morphic halo)

그림 1.3: 월드 메뉴 (빨강 마우스 버튼으로 불러옴), 상황 메뉴 (노랑 마우스 버튼), 그리고 모픽 halo (파랑 마우스 버튼).

스퀵은 "왼쪽 마우스 버튼 클릭"과 같은 용어 사용을 피합니다. 그 이유는, 다양한 컴퓨터, 마우스, 키보드 및 개인 설정은, 다양한 사용자들이 동일한 효과를 내기 위해 다양한 물리적 버튼들을 눌러야 할 필요를 의미하기 때문입니다. 그 대신에, 마우스 버튼들에 색상을 입혀 라벨을 붙였습니다. 월드 메뉴를 얻기 위해 여러분이 누른 마우스 버튼은 빨강 버튼 으로 불리며, 그 빨강 버튼은 목록들에 있는 항목 선택, 텍스트 선택, 그리고 메뉴 항목 선택을 하기 위해 가장 빈번하게 사용됩니다. 여러분이 스퀵사용을 시작할 때, 그림 1.4 에 보이는 것처럼 마우스에 실제로 라벨을 붙이면 상당한 도움이 될 수 있습니다.[2]

노랑 버튼은 그 다음으로 가장 많이 사용하는 버튼이며, 마우스가 가리키는 위치에 다양한 동작의 모음을 제공하는 메뉴인 상황 메뉴를 불러올 때 사용합니다. 그림 1.3b 를 보시기 바랍니다.

[2] "빨강 클릭"이라는 용어 사용을 피하고, 대신 기본적으로 "클릭"을 사용하겠습니다.

*Workspace*에 `Time now`를 입력 하십시오. 이제 *Workspace*에서 지금 노랑색 버튼을 클릭하고 print it을 선택하십시오.

마지막으로, 객체들의 회전, 크기 조정과 같이 화면 상의 객체들에 대한 동작들을 수행하는 작업에 사용하는 손잡이 배열인 "모픽 할로"를 활성화 하는 파랑 버튼이 있습니다. 그림 1.3c 를 참조하십시오. 여러분이 손잡이 위에 마우스를 올려놓으면, 핸들의 기능을 설명하는 도움말 풍선을 띄웁니다.

Workspace 위에서 파랑 버튼을 클릭하십시오. *Workspace*를 회전하려면 하단 왼쪽 모서리 근처의 손잡이를 잡고 끌어 이동하십시오.

오른손잡이 사용자들께서는 마우스의 왼쪽에 빨강 버튼, 오른쪽에 노랑색으로 지정하시고, 클릭 할 수 있는 스크롤 휠이 있다면 파랑 버튼 지정을 권합니다. 클릭 할 수 있는 스크롤 휠이 없다면, 빨강 버튼을 누르고 있는 동안, alt 또는 option 키를 함께 클릭하여 "모픽 할로"를 볼 수 있습니다. 두 번째 버튼이 없는 매킨토시 컴퓨터를 사용하신다면, 마우스 버튼을 클릭하는 동안 ⌘ 키를 함께 누른 상태를 유지하여 두 번째 버튼 동작을 흉내낼 수 있습니다. 그럼에도 불구하고, 여러분이 스퀵을 자주 사용하시려 한다면, 적어도 2개 이상의 버튼을 가진 마우스 구입에 투자하시기를 권합니다.

여러분은 운영체제와 마우스 드라이버의 기본 설정을 사용하여 원하는 방식으로 마우스가 동작하도록 구성할 수 있습니다. 스퀵은 마우스와 여러분의 키보드에 있는 메타키를 사용자 지정하기 위한 몇 가지 기본 설정 요소를 갖고 있습니다. 여러분은 World 메뉴의 open 항목에서 기본 설정 브라우저를 찾을 수 있습니다. 기본 설정 브라우저에서, general 카테고리에서는 노랑 및 파랑 기능을 바꾸는 swapMouseButtons 옵션을 가지고 있습니다 (그림 1.5 참조). keyboard 카테고리는 다양한 명령 키를 복사하기 위한 옵션들을 갖고 있습니다.

기본 설정 브라우저를 열고, 검색 상자를 사용하여 swapMouseButtons 옵션을 찾으십시오.

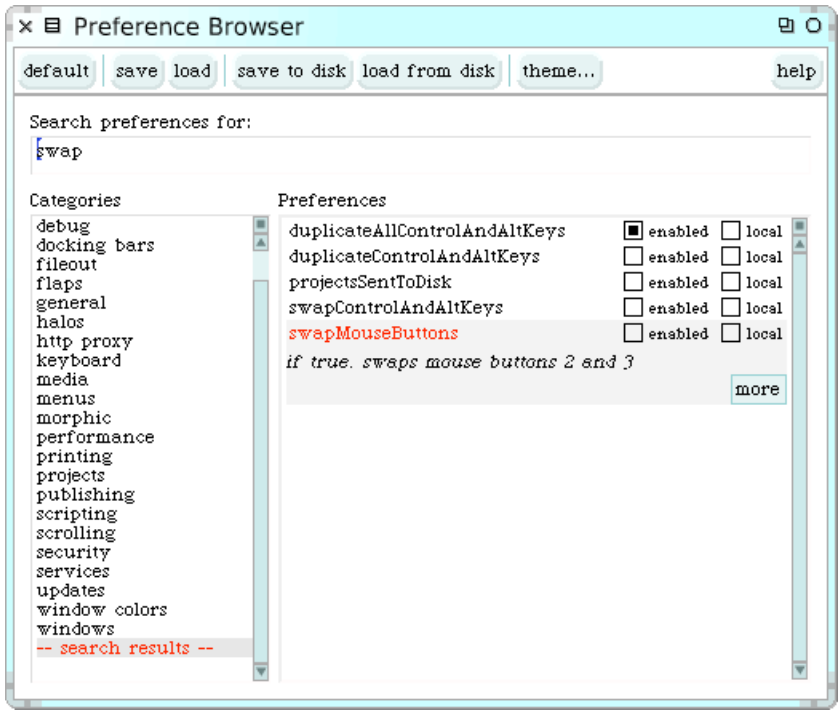


그림 1.5: 기본 설정 브라우저(The Preference Browser)

1.2 world 메뉴

스퀵배경을 다시 클릭하십시오.

World 메뉴를 다시 보게 될 것입니다. 대부분의 스퀵메뉴는 "modal(상태적 요소)"이 아니며, 여러분은 그 메뉴들을 상단 우측 모서리에 있는 "push pin(고정하기)" 아이콘을 클릭하여 원하는 시간동안 화면에 남겨둘 수 있습니다.

시도해보세요. 사용자가 마우스를 클릭하면 메뉴가 나타나지만, 그 메뉴를 놓을 때 사라지지 않음을 알아두시기 바랍니다. 사용자가 선택을 하거나 메뉴의 바깥부분을 클릭할 때까지는 보이는 상태로 머무르게 됩니다. 사용자는 메뉴의 제목 표시줄을 잡고 메뉴를 주변으로 움직일 수도 있습니다.

World 메뉴는 스퀵이 제공하는 많은 도구들에 접근하기 위한 단순한 수단을 제공합니다.

 World ▷ open… 메뉴를 자세히 살펴 보십시오

스퀵에서 시스템 탐색기(사용가능한 많은 클래스 탐색기중 하나)와 워크스페이스를 포함한 여러 개의 핵심 도구 목록을 보실 것입니다. 독자는 앞으로 볼 내용에서 이러한 대부분의 도구를 접하게 됩니다.

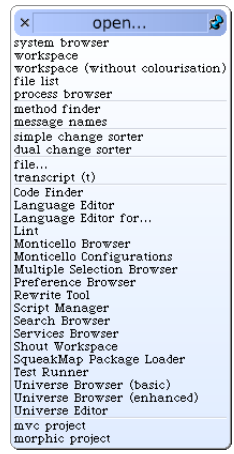


그림 1.6: 월드 메뉴의 open … 대화상자

1.3 스퀵세션 저장하기, 그만두기 다시 시작하기

 월드 메뉴를 불러오십시오. new morph를 선택하고, alphabetical list ▷ A-C ▷ BlobMorph 순으로 탐색하십시오. 이제 여러분은 *"손에" blob*을 넣게 되었습니다. *blob*을 *(클릭하여)* 어딘가에 놓아두십시오. 옮기기를 시작합니다.

 World- ▷ save as… 를 선택하고 이름 *"SBE"* 를 입력하십시오. 이제 Accept (s) 버튼을 클릭하십시오. 이제 World ▷ save and quit 를 선택하십시오.

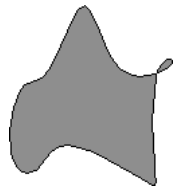


그림 1.7: BlobMorph 인스턴스.

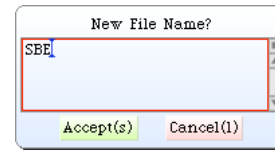


그림 1.8: Save as… (다른 이름으로 저장) 대화상자.

이제 여러분이 원본 image 파일과 changes 파일이 있는 위치로 이동하면, save and quit 를 실행하라고 알리기 전에 스퀵이미지의 작업 상태를 보여주는 두 개의 새로운 파일 "SBE.image 와 *"SBE.changes* 를 보실 수 있습니다. 원하신다면, 디스크의 어느곳에라도 두 개의 파일들을 옮길 수 있지만, 그렇게 옮기려면, 가상 머신과 *.source*파일에 두 개의 파일을 이동하고 복사하고 링크할 필요가 있습니다.

새롭게 만들어진 *"SBE.image"* 파일에서 스퀵을 시작하십시오.

이제 스퀵에서 나갈 때의 상태를 정확하게 찾아야만 합니다. "blob" 은 이전에 스퀵에서 나갈 때의 상태에서 다시 시작하고, "blob" 작업을 그만두었을 때의 위치에서 다시 이동을 계속합니다.

처음 스퀵을 시작할 때, 스퀵가상 머신은 여러분이 제공하는 image 파일을 불러옵니다. 이 파일은 이전에 존재하는 수많은 양의 코드와 다수의 프로그래밍 도구(이 모든 것은 오브젝트입니다)를 포함한 다수 객체의 스냅샷을 포함하고 있습니다. 스퀵으로 작업을 할 때, 이러한 객체들에 메시지를 보내고, 새로운 객체들을 만들 것이며, 이 객체중 일부는 수명을 다하여 없어지고 그 오브젝트의 메모리를 되돌릴 것입니다. (예, 가비지 수집)

스퀵을 끝낼 때, 보통 모든 객체를 포함한 스냅샷을 저장할 것입니다. 일상적인 저장을 하면, 새로운 스냅샷과 함께 옛 image 파일을 덮어쓸 것입니다. 대신 방금 해보았던대로 새로운 이름으로 이미지를 저장할 수도 있습니다.

.image 파일 뿐만 아니라, .changes 파일도 있습니다. 이 파일은 표준 도구를 사용하여 만든 소스 코드에 모든 변경 사항의 기록을 포함합니다. 대부분의 경우 이 파일에 대해 여러분이 걱정할 필요는 전혀 없습니다. 그러나 우리가

보게 될 것은, *.changes* 파일은 에러를 복구하거나 잃어버린 변경 사항을 다시 실행하는 작업에 매우 유용하다는 점입니다. 이 내용에 대한 좀 더 많은 부분은 나중에 살펴보겠습니다!

여러분이 함께 작업을 해온 이미지는 1970년대 후반에 만들어진 원본 "smalltalk-80 image" 의 후예들입니다. 어떤 객체는 (만들어진 지) 수 십 년이 지나기도 했습니다!

아마도, 이미지는 소프트웨어 프로젝트를 저장하고 관리하는 핵심 기술이라고 생각하시겠지만, 사실은 그렇지 않습니다. 곧 보게 될 내용처럼, 팀이 개발한 코드를 관리하고 소프트웨어를 공유하기 위한 훨씬 더 나은 도구들이 있습니다. 이 이미지들은 매우 유용합니만, 몬티첼로*Monticello* 와 같은 도구들이 개발자들간에 버전을 관리하고 코드를 공유하는 훨씬 나은 방법들을 제공하기 때문에 이미지를 만들고 버리는 것에 대해 매우 무관심 해지는 법을 배우실 필요가 있습니다.

 *blob*에서 파랑-클릭을 하십시오.

"blob" 의 모픽할로*Morphic halo* 라 하는 색칠한 점의 모임을 보실 것입니다. 각 점은 핸들이라고 합니다. (하지만 앞으로 손잡이라고 하겠습니다. 역자 주.) 십자 모양을 포함하고 있는 분홍색 손잡이를 클릭하면, "blob" 이 사라집니다. (blob이 주변을 꿈틀거리면 여러번 클릭을 해야 할 수도 있으며, 마우스로부터 벗어나기를 시도합니다.)

1.4 Workspace와 Transcript

 열려 있는 모든 창을 닫으십시오. 도구 플랩*(the Tools flap)*을 열기 위해 스퀵환경 맨 오른쪽에 있는 Tools 탭을 클릭하십시오.

스퀵의 몇 가지 핵심 도구를 위한 아이콘(그림 1.9)을 보실 것입니다. Transcript와 Workspace 을 끌어 움직입니다.

 *Transcript*와 *Workspace* 창의 위치와 크기를 조절하여, *Workspace*와 *Transcript*를 정확히 겹치도록 만드십시오.

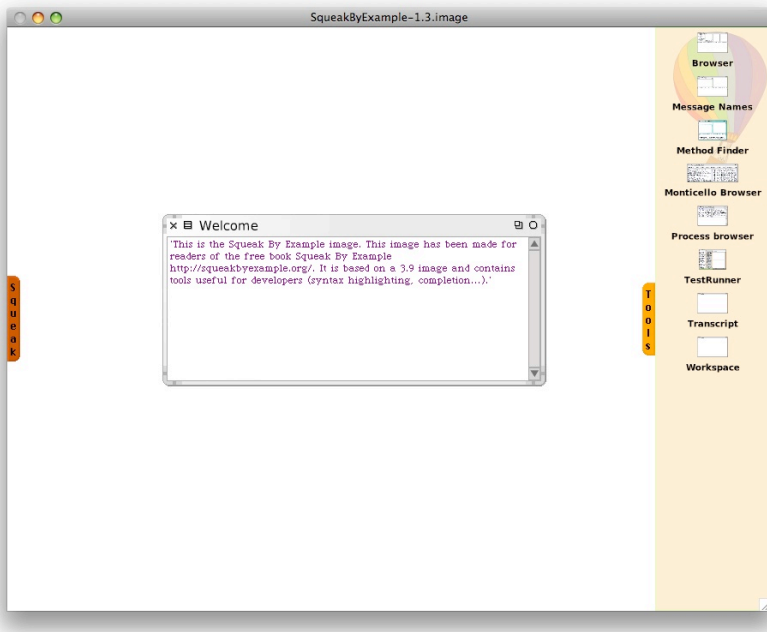


그림 1.9: 스퀵*Tools* 플랩.

모서리들 중 한 곳을 끌어 움직이거나, "모픽 손잡이morphic handles"를 불러오기 위해 창을 파랑 클릭하고 하단 아래쪽의 노랑 핸들을 끌어 움직여 창 크기를 다시 조정 할 수 있습니다.

언제든 오직 한 개의 창만 활성화됩니다. 항상 위에 위치하고 해당 라벨이 강조됩니다. 마우스 커서는 입력을 원하는 창에 반드시 위치해야 합니다.

Transcript는 시스템 메시지를 기록하기 위해 종종 사용하는 객체입니다. Transcript는 "시스템 콘솔"의 한 종류입니다. Transcript는 매우 느리므로, Transcript를 열고 특정 연산을 위해 기록할 경우, 10배나 더 느려진다는 것을 참고하셔야 합니다. 뿐만 아니라, Transcript는 스레드로부터 안전하지 않으므로, 여러 객체를 Transcript에 동시에 기록할 경우 이상한 문제들을 경험할 수 있습니다.

Workspace는 여러분이 실험하고 싶은 스몰토크코드 일부를 입력하는 작업에 유용합니다. 여러분은 또한 단순히 할 일 목록*to-do lists* 또는 누군가가 여러분의 이미지를 사용할 때를 위한 설명서와 같이, 기억을 원하는 임시 텍스트를 입력하는 용도로 Workspace 를 사용할 수 있습니다. Workspace는 이전에 다운로드한 표준 이미지의 경우와 같이, 종종 캡쳐한 이미지에 대한 문서를 유지하기 위해 자주 사용합니다. (그림 1.2 참조)

 다음 텍스트를 *Workspace*Workspace 에 입력하십시오.

```
Transcript show: 'hello world'; cr.
```

방금 입력한 텍스트의 아무 지점에서 Workspace*Workspace* 더블 클릭 해보십시오. 클릭 장소에 따라, 어떻게 전체 단어, 전체 문자열 또는 전체 텍스트를 선택하는지에 주목하도록 합니다.

 입력한 텍스트를 선택하고 노랑 클릭하십시오. do it (d) 을 선택하십시오.

Transcript창 (그림 1.2) 에 "hello world" 텍스트가 어떻게 나오는지 살펴보십시오. 다시 해보십시오. (메뉴 항목 do it (d) 에 있는 (d) 는 단축키가 CMD−d 임을 알려줍니다. 더 많은 내용은 다음 장을 참고하십시오!)

방금 첫번째 스몰토크프로그램식을 처리했습니다! Transcript 객체에 메시지와
cr[개행 문자(carrige return)]가 따라오는 show: 'hello world' 메시지를 보냈습니다. Transcript는 이후 메시지로 무엇을 해야 할 지 결정하고 show:와 cr 메시지를 다루기 위한 메서드를 찾았으며, 적절하게 반응하였습니다.

스몰토커와 한동안 대화를 해보셨다면, "연산 호출 (call an operation)" 또는 "메서드 불러오기 (invoke a method)" 와 같은 표현을 사용하지 않고, 그 대신 "메시지 보내기 (send a message)" 라는 표현을 사용함을 재빨리 눈치채셨을 것입니다. 객체가 그 자체의 동작에 책임을 갖고 있다는 생각을 반영합니다. 객체가 무슨 일을 하는지 *알려* 주는 것이 아니라, 대신 메시지를 보내어 객체에게 무엇을 할 지 정중하게 *요청*합니다. 여러분이 아닌 객체가 여러분의 메시지에 반응하기 위한 적절한 메서드를 선택합니다.

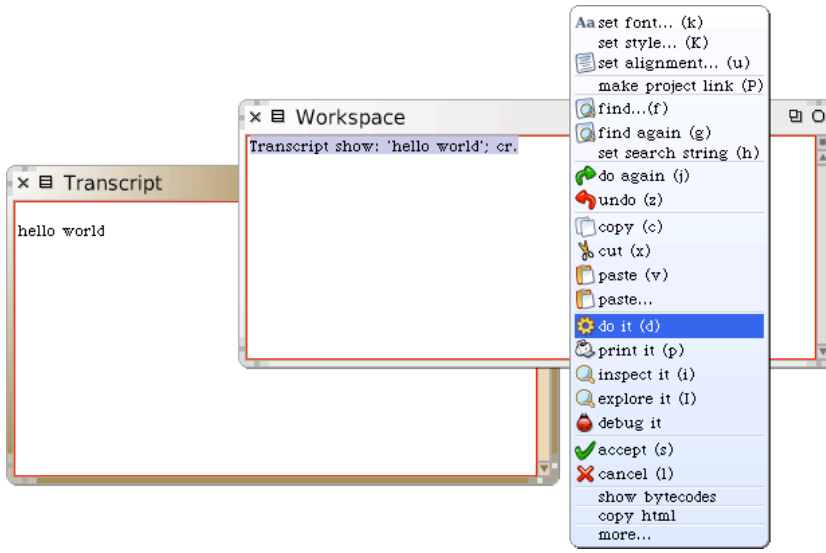


그림 1.10: 프로그램식 "수행"

1.5 키보드 단축키

표현식을 처리하기 위해, 항상 노랑 버튼 메뉴를 불러올 필요는 없습니다. 그 대신, 키보드 단축키를 사용할 수 있습니다.

메뉴에 괄호로 묶은 표현이 있습니다. 사용자가 보유한 플랫폼에 따라, 수정자 키(`modifier key`, `control`, `alt`, `command` 또는 `meta`키. `meta`키는 `super`키라고도 부릅니다.)중 하나를 눌러야 합니다. (우리는 이 키들을 CMD–KEY라고 하겠습니다.)

 *Workspace* 창에서 구문을 다시 처리해보십시오. 대신 키보드 단축키 *CMD–d* 를 사용하십시오.

`do it` 뿐만 아니라 `print it`, `inspect it`, `explore it` 도 보실 것입니다. 이들 각각을 잠시 각각 살펴보겠습니다.

 계산식 `3+4`를 *Workspace*에 타이핑하십시오. 이제 키보드 단축키로 `do it`을 실행하십시오.

아무 일도 일어나지 않는 것에 놀라지 마십시오! 방금 하신 일은 인자 4과 + 메시지를 숫자 3에 보냈을 뿐입니다. 보통 7이라는 결과로 처리하여 되돌려주지만, Workspace가 이 답을 갖고 무엇을 해야 할 지 모르기 때문에, 그 Workspace는 단순히 이 답을 내보냈습니다.

만약 결과를 보시려면, 대신 print it 을 실행하셔야 합니다. print it 은 실제로 구문에 대한 컴파일 작업과 실행 작업을 한 후 printString 메시지를 결과로 내보내며, 결과 문자열을 표시합니다.

 3+4를 선택하고 print it *(CMD−P)* 을 실행하십시오

이번에는 기대하던 결과를 볼 수 있습니다. (그림 1.11)

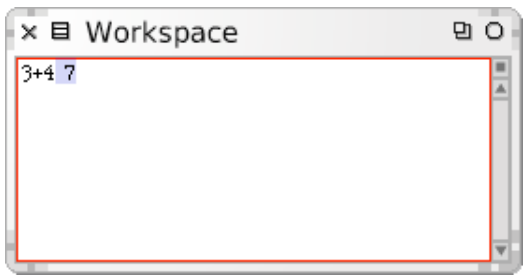


그림 1.11: "do it" 보다는 "print it" 을 실행하십시오

```
3 + 4  ⟶  7
```

이 책에서 --> 기호는 print it 을 실행할 때, 주어진 결과를 내보내는 특정 스퀵 계산식의 일부를 나타내는 약속입니다.

 강조된 텍스트 7을 지우십시오 *(*스퀵은 선택을 해야 *delete* 키를 누를 수 있게 해줍니다*)*. 3+4를 다시 선택하고 이번에는 inspect it *(CMD−i)* 을 실행하십시오.

이제 `SmallInteger:7` (그림 1.12) 이라는 제목을 가진 Inspector 라는 새로운 창을 보실 것입니다. 이 Inspector 창은 시스템의 모든 객체를 탐색하고 상호작용할 수 있게 하는 굉장히 유용한 도구입니다. 제목에서는 7이 SmallInteger 클래스의 인스턴스라는 점을 알려줍니다.

왼쪽 패널은 객체의 인스턴스 변수를 검색할 수 있게 해주며, 그 변수의 값은 오른쪽 패널에 나타납니다. 아래의 패널은 메시지를 객체에 보내기 위해 표현식을 쓰는 용도로 사용할 수 있습니다.

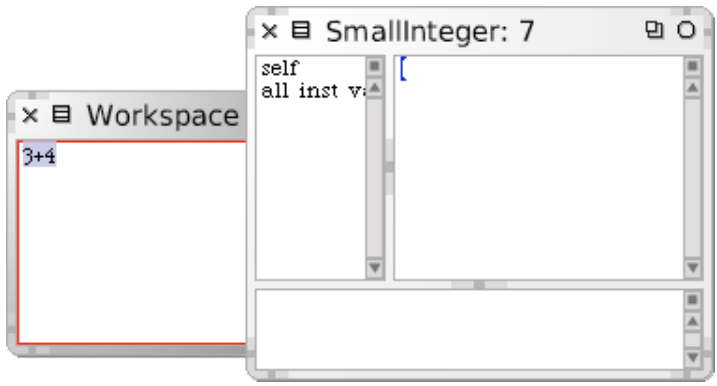


그림 1.12: 객체의 검사

 *7*이 있는 *Inspector* 창의 하단 패널에 `self squared`를 입력하고 Print it 을 실행하십시오.

 *Inspector* 를 닫으십시오. *Workspace* 창에 `Object` 구문을 입력하고 이번에는 explore it (CMD–I, 대문자 *i* 입니다)을 실행하십시오.

이번에는 Object 라고 라벨이 붙은 ▷ `root: Object` 텍스트가 들어있는 창을 보실 것입니다. 창을 펼치기 위해 삼각형 모양(그림 1.13)을 클릭하십시오.

이 탐색기는 Inspector 창과 유사하지만, 복잡한 객체의 모양새를 트리 모양으로 보여줍니다. 지금의 경우 우리가 보고 있는 객체는 `Object` 클래스입니다. 이 클래스에 저장된 모든 정보를 직접 볼 수 있으며, 모든 그 정보의 부분들을 돌아다니며 쉽게 찾아볼 수 있습니다.

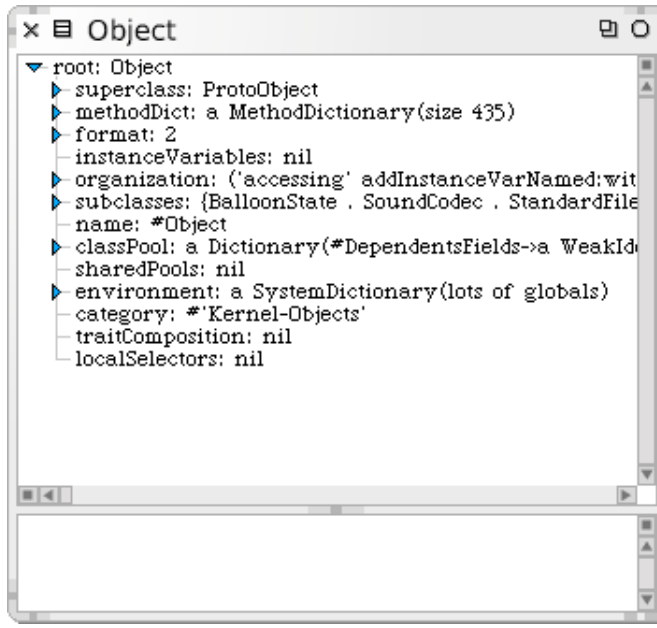


그림 1.13: 객체의 탐색.

1.6 SqueakMap

SquakMap은 여러분의 이미지에 다운로드 할 수 있는 웹 기반의 "패키지"-프로그램 라이브러리 모음-의 웹 기반 카탈로그입니다. 패키지는 전 세계의 다양한 곳에서 제공하며, 수 많은 사람들이 관리합니다. 패키지 중 일부는 스퀵의 일부 버전에서만 동작합니다.

 World ▷ open⋯ ▷ SqueakMap package loader 를 여십시오.

이 작업을 수행 하려면 인터넷 연결이 필요합니다. 얼마 후, SqueakMap 로더 창이 나타날 것입니다 (그림 1.14). 왼쪽 측면에는 매우 긴 패키지 목록이 나타납니다. 상단 왼쪽 모서리 필드는 목록에서 원하는 것을 찾아보기 위한 검색 창 입니다. 검색 창에 "Sokoban" 을 타이핑하고 return 키를 치십시오. 사용 가능한 버전을 나타내는 패키지의 이름을 오른쪽으로 가리키는 삼각형을 클릭해보십시오. 사용가능한 패키지와 버전을 선택하면, 선택 항목에

대한 정보가 오른쪽 패널에 나타납니다. Sokoban의 최종 버전 내용을 탐색하십시오. 목록 창에서, 선택한 패키지를 `install` 하기 위해 노랑 버튼을 사용하십시오(스퀵이 이 게임의 버전이 여러분의 이미지에서 실행될지 확실하지 않다는 메시지를 보내면, 그냥 "Yes"를 누르고 진행해 주십시오). 일단 패키지를 설치하면, SqueakMap 패키지 로더의 목록에서 *(별표)로 표시됨을 확인하십시오.

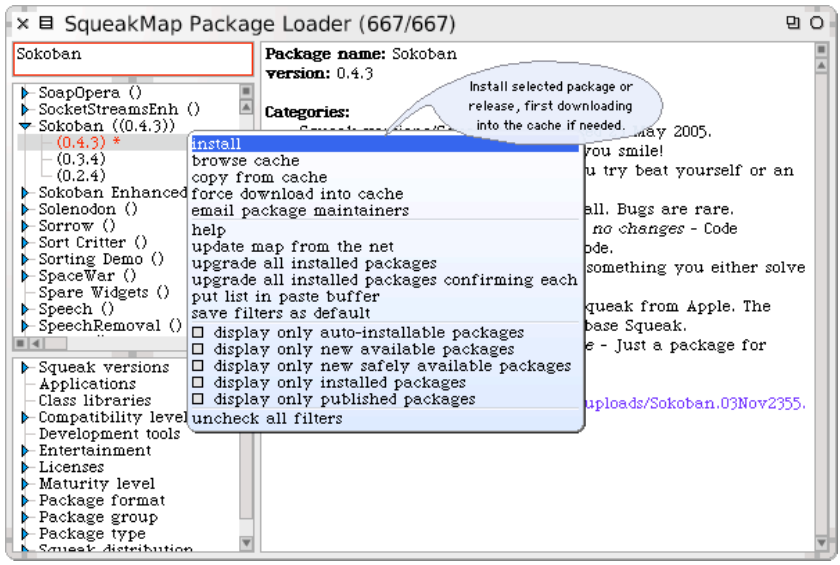


그림 1.14: Sokoban 게임을 설치하기 위해 SqueakMap 활용하기.

 이 패키지를 설치 한 후, *Workspace* 창에서 `SokobanMorph random openInWorld` 를 실행하여 `Sokoban`을 시작하십시오.

SqueakMap 패키지 로더의 하단 왼쪽 창은 특정 해당항목을 표시하는 다양한 방법을 제공합니다. 스퀵의 특정 버전과 호환 가능한지, 혹은 단지 게임하고만 호환이 가능한지 등을 알아보기 위해 선택해볼 수 있습니다.

1.7 시스템 브라우저 (The System Browser)

시스템 브라우저는 프로그래밍에 사용하는 핵심 도구중 하나입니다. 우리가 앞으로 살펴볼 내용처럼, 스퀵에서 사용할 수 있는 여러가지 흥미로운 브라우저가 있지만, 시스템 브라우저는 어떤 이미지에서든 찾으실수 있는 기본 브라우저 입니다.

 World ▷ Open⋯ ▷ Class browser 순서로 선택하시거나 *Tools* 플랩 에서 브라우저를 끌어 옮기십시오.

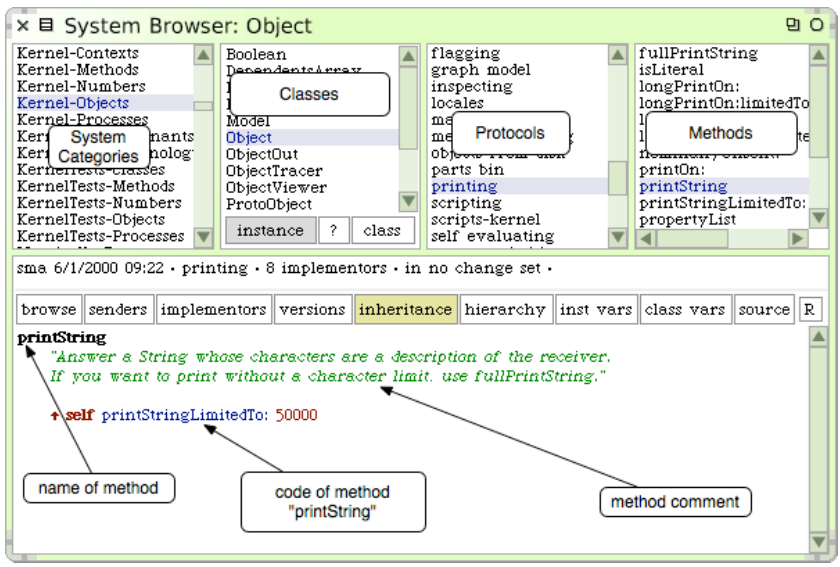


그림 1.15: Object 클래스의 prinsString 메서드를 보여주는 시스템 브라우저

그림 1.15 에서 시스템 브라우저를 볼 수 있습니다. 제목 표시줄은 사용자가 검색하는 Object 클래스를 나타냅니다.[3]

브라우저를 처음 열면 가장 왼편을 제외한 나머지 창은 비어 있습니다. 이 첫 번째 창에서는 알려진 모든 관련 클래스의 그룹 시스템 카테고리를 나열합니

[3]이 책에서 설명한 것과 브라우저의 모습이 다르다면 다른 기본 브라우저를 사용하는 것일 수도 있습니다. P.272의 FAQ 5번을 보십시오

다.

 `Kernel-Objects` 카테고리를 클릭하십시오.

이 동작은 선택한 카테고리에 있는 모든 클래스의 목록을 보여주는 두 번째 창을 만듭니다.

 `Object` 클래스를 선택하십시오.

이제, 남은 두 개의 창을 텍스트로 채울 것입니다. 세번째 창은 현재 선택된 클래스의 프로토콜을 표시합니다. 이들이 잘 모아놓은 메서드입니다. 선택한 프로토콜이 없다면, 네번째 창에서 모든 메서드를 볼 수 있습니다.

 `printing` 프로트콜을 선택하십시오.

이 프로토콜을 선택하려면 스크롤을 내려봐야 합니다. 이제 화면 출력 동작과 관련된 유일한 프로토콜을 네번째 창에서 볼 수 있습니다.

 `printString` 메서드를 선택하십시오.

이제 하단 창에서, 시스템의 모든 객체(오버라이드 제외)가 공유하는, printString 메서드의 소스 코드를 볼 수 있습니다.

1.8 클래스 찾기

스퀵에서 클래스를 찾는 여러가지 방법이 있습니다. 우선 첫째로, 앞에서 보신 바와 같이, 어떤 카테고리에 클래스가 있는지를 알아내고(또는 추측하고), 브라우저를 사용하여 탐색하는 것입니다.

두 번째 방법은 클래스에 browse 메시지를 보내어 자체적으로 브라우저를 열도록 요청하는 것입니다. `Boolean` 클래스를 탐색하려는 상황을 가정해보겠습니다.

 워크스페이스에 `Boolean` *browse* 를 입력하고 do it을 실행하십시오.

브라우저는 Boolean 클래스를 엽니다(그림 1.16). 클래스 이름을 찾은 어떤 도구에서든 키보드 단축키 CMD–b (browse) 도 존재합니다. 이름을 선택하고 CMD–b 를 입력하십시오

 Boolean 클래스를 검색하기 위해 키보드 단축키를 사용하십시오.

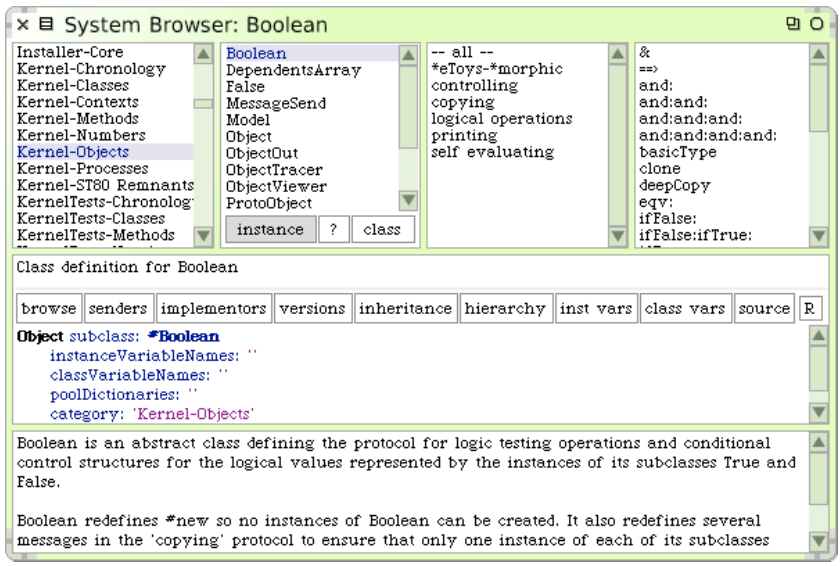


그림 1.16: Boolean 클래스 정의를 보여주는 시스템 브라우저

Boolean 클래스를 선택했지만, 선택한 프로토콜과 메서드가 없다면, 상단에 네 개의 패널 밑에 하나의 패널이 아닌 두 개의 패널이 나타남을 확인하십시오(그림 1.16). 위의 패널은 클래스 정의 내용이 들어있습니다. 서브클래스 만들도록 요청하여 super클래스에 보낸 일반적인 스몰토크메시지일 뿐입니다. 여기서 Object 클래스가 인스턴스 변수, 클래스 변수 또는 "pool Dictionary" 를 갖지 않는 Boolean 서브클래스 생성과 *Kernel-Objects* 카테고리로의 배치를 요청 받았음을 보고 있습니다.

하단 창에서는 클래스 주석 — 클래스를 설명하는 일반텍스트*plain text* 부분을 보여줍니다. 클래스 창의 하단에서 ? 버튼을 클릭하면, 보고 있는 창에서 클래스 주석을 볼 수 있을 것입니다.

여러분이 스퀵의 상속 계층을 탐색한다면, 계층 브라우저가 도와드릴 수 있습니다. 여러분이 알려진 클래스의 알려지지 않은 서브클래스 또는 super클래스를 찾을 때 이 브라우저를 유용하게 사용할 수 있습니다. 계층 브라우저는 상속 계층를 미러링하는 들쭉날쭉한 트리구조로 나열합니다.

 `Boolean`클래스를 선택하고, 브라우저에 있는 hierarchy를 선택하십시오.

이 작업은 `Boolean` 클래스의 서브클래스와 super클래스를 보여주는 계층 브라우저를 엽니다. `Boolean`의 인스턴트 상위클래스와 서브클래스를 바로 탐색해보십시오.

자주 클래스를 찾는 가장 빠른 방법은 그 클래스의 이름으로 검색하는 것입니다. 예를 들어,, 날짜와 시간을 나타내는 알려지지 않은 클래스를 찾고 있다고 가정해 보겠습니다.

 시스템 브라우저의 시스템 카테고리 창에 마우스를 올려놓고 *cmd–f*를 누르거나, 노랑 버튼 메뉴에서 find class… (f) 를 선택하십시오. 대화 상자에 *"time"*을 입력하고 *accept* 하십시오.

"time"을 포함하는 이름을 가진 클래스의 목록이 나타날 것입니다. (그림 1.17을 보십시오) 한 가지를 고른 후, Time을 입력하면, 브라우저는, 쓸만한 다른 클래스를 제안하는 클래스 주석과 함께, 목록을 보여드립니다. 여러분이 다른 클래스의 목록 중 하나를 검색하려면, 그 클래스의 이름을 선택하고 (아무 텍스트 패널에서) cmd–b 를 입력하십시오.

찾기 대화상자에서 클래스의 전체 이름을(정확하게 대소문자 구별) 입력하였다면, 브라우저는 옵션의 목록을 보여주지 않고 입력한 클래스로 바로 이동합니다.

1.9 메서드 찾기

때로는, 여러분이 메서드의 이름을 추측할 수 있거나, 클래스의 이름보다는 적어도 메서드 이름의 최소한의 부분을 보다 더 쉽게 추측 할 수 있습니다.

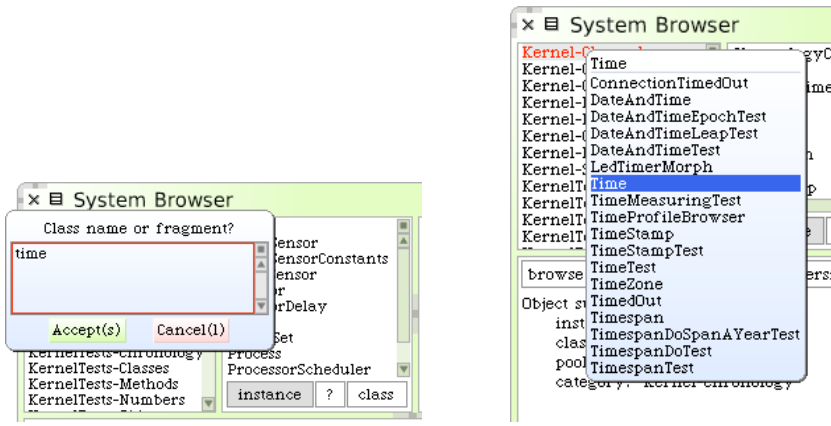


그림 1.17: 이름으로 클래스 검색하기

예를 들어,, 만약 현재 시간이 궁금할 경우, "now"라 불리는 메서드나 하위 문자열에 "now"를 포함하는 메서드를 기대할 것입니다. 그런데 이 메서드가 어디에 있을까요? 메서드 파인더(the method finder)가 도와드릴 수 있습니다.

Tools flap" 밖으로 메서드 파인더 아이콘을 끌어내십시오. 상단 왼쪽 창에 "now"를 입력하고 accept 하십시오. (또는 그냥 return 키를 누르십시오.)

메서드 파인더는 하위 문자열 "now"를 포함하고 있는 모든 메서드 이름의 목록을 표시할 것입니다. "now"를 향해 스크롤을 내리려면, 목록으로 커서를 옮기고 "n"을 입력합니다. 이 기술은 모든 스크롤 창에서 동작합니다. "now"를 선택하면 그림 1.18 에서 보시는 바와 같이 오른쪽 창에서 이 이름으로 정의한 세가지 메서드를 보여줍니다. 이 항목 중 아무거나 한가지를 선택하면 해당 창에 브라우저를 엽니다.

평소에는 메서드가 존재한다는 바람직한 생각을 갖고 계시겠지만, 그것이 어떤 이름으로 불리는 지에 대한 생각은 하고 있지 못하실 것입니다. 이 상황에서도 메서드 파인더는 도움가 도움을 줄 수 있습니다! 예를 들어, `'eureka'`를 `'EUREKA'`로 모든 문자를 대문자로 변환하는 상황을 가정해봅시다.

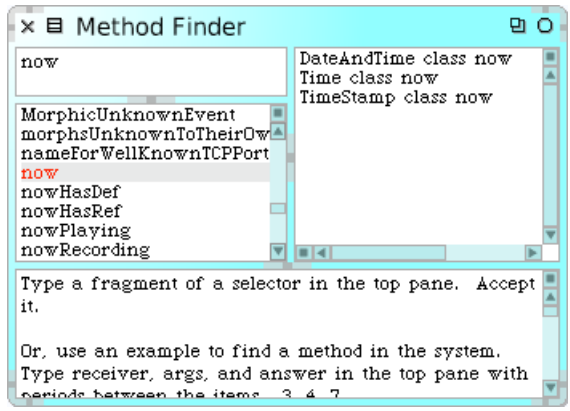


그림 1.18: now 이름을 가진 메서드를 정의한 세가지 클래스를 보여주는 메서드 파인더.

그림 *1.19* 에서 보이는 것처럼 `'eureka'.'EUREKA'`를 메서드 검색에 입력하고 *RETURN*키를 누르십시오.

메서드 파인더는 여러분이 원하는 메서드를 제시할 것입니다. 메서드 파인더의 우측 창 줄의 시작부분에 있는 *(별표) 문자는 요청한 결과를 가져오기 위해 실제로 사용한 메서드임을 가리킵니다. 따라서, String asUppercase의 앞 부분의 *(별표) 문자는, *String* 클래스에 정의된 asUppercase 메서드를 실행했으며, 우리가 원했던 결과를 반환했음을 의미합니다. *(별표)를 갖고 있지 않은 메서드는 기대했던 결과를 반환하는 메서드와 동일한 이름을 가진 다른 메서드입니다. 따라서 'eureka'가 Character 오브젝트가 아니므로 `Characater>>asUppercase` 을 실행하지 않았습니다.

또한 인자를 가진 메서드에 대해서도 메서드 파인더를 사용할 수 있습니다. 예를 들어, 만약 여러분이 두 개의 정수의 최대 공약수를 찾는 메서드를 찾고 있다면, 그 예로 25. 35. 5를 시도해볼 수 있습니다. 또한 탐색 범위를 좁히기 위해 메서드 검색에 여러 가지 예를 제시할 수 있으며, 하단 창에 있는 도움말 텍스트로 더 많은 내용을 설명합니다.

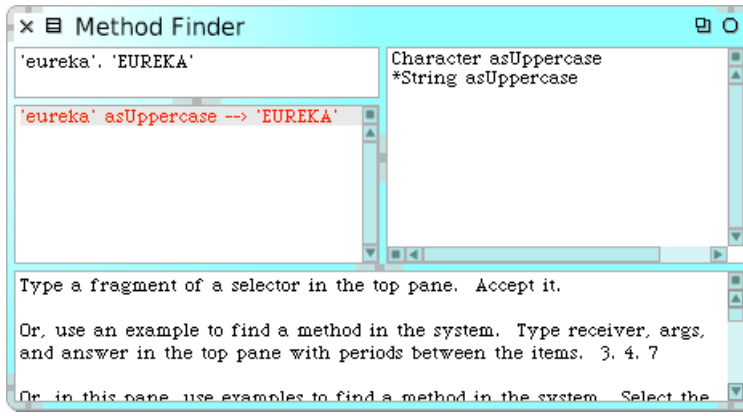


그림 1.19: 메서드 찾기 예제

1.10 새로운 메서드 정의하기

테스트 주도 개발 (Test Driven Development) [2]의 출현은 코드를 작성하는 방식을 바꾸었습니다. TDD 또는 행동 주도 개발 (Behavior Driven Development)이라고 하는 테스트 주도 개발 (Test Driven Development)의 배경 개념은, 코드 자체를 작성하기 전에, 원하는 코드 동작을 정의하는 테스트를 작성하는 것에 있습니다. 그 다음, 테스트를 만족하는 코드를 작성하는거죠.

이제부터 해야할일이 "무엇인가를 강조하여 크게 말하기"[4] 를 수행할 메서드를 만드는것이라고 가정해봅시다. 이건 정확하게 어떤 의미가 될까요? 메서드에 붙일 좋은 이름은 무엇일까요? 무엇을 해야 할 지 모호한 동작 설명을 가지도록 나중에 메서드를 관리해야 하는 프로그래머들에게 어떻게 이해시킬 수 있을까요? 이런 질문들에 대한 답은 아래와 같습니다.

> `shout`메시지를 문자열 "Don't panic" 에 보내면, 그 결과는 "DON'T PANIC!" 이 되어야 합니다.

[4] 원문은 `say something loudly and with emphasis`

시스템이 사용할 수 있는 무언가에 이 예시를 만들려면, 이 예를 test 메서드로 변환해야 합니다:

Method 1.1: `shout` 메서드를 위한 테스트

```
assert: ('Don''t panic' shout = 'DON''T PANIC!')
```

스퀵에서는 어떻게 이 새로운 메서드를 만들 수 있을까요? 먼저 어떤 클래스에 메서드를 넣을 것인지 결정해야 합니다. 이 경우, String 클래스에 테스트할 `shout` 메서드를 넣고, 관례적으로 이 클래스에서 실행할 관련 테스트 클래스를 StringTest 라고 하겠습니다.

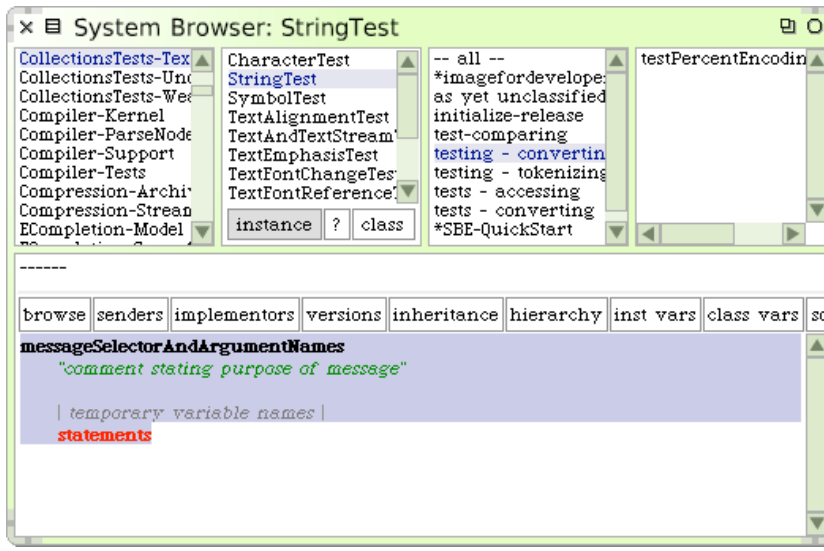


그림 1.20: `StringTest` 클래스의 새 메서드 템플릿.

 `StringTest` 클래스에서 브라우저를 열고, 메서드에 적합한 프로토콜을 선택하십시오. 이 같은 경우 그림 *1.20* 에서 보시는 바와 같이 tests - converting 이 적당한 프로토콜입니다. 하단 창의 강조된 텍스트는 스몰토크메서드가 어떻게 생겼는지 알려주는 템플릿입니다. 이 부분을 지우고 메서드 *1.1* 에 코드를 입력하십시오.

브라우저에 텍스트를 입력하고 나면, 하단 창이 빨간 테두리로 둘러 쳐짐을 보십시오. 이는 저장하지 않은 내용이 창에 있음을 알려주는 알리미입니다. 따라서, 여러분이 작성하던 메서드를 컴파일하고 저장하려면 하단 창의 노랑 버튼 메뉴에서 accept(s) 를 선택하거나, CMD−S 를 입력하십시오.

아직 shout라고 하는 메서드가 없기 때문에, 브라우저에서 이 이름을 정말 사용할 것인지 물어보고 여러분이 의도하는 다른 이름들을 제안할 것입니다 (그림 1.21). 이러한 동작은 입력을 실수했을 때 약간 쓸모있을 수도 있지만, 이번과 같은 경우에는 정말로 만들고자 하는 shout 메서드를 *실행*하려는 의미이므로, 그림 1.21 에서 보시는 바와 같이 메뉴 선택으로부터 첫번째 옵션을 선택하여 이를 확인해야 합니다.

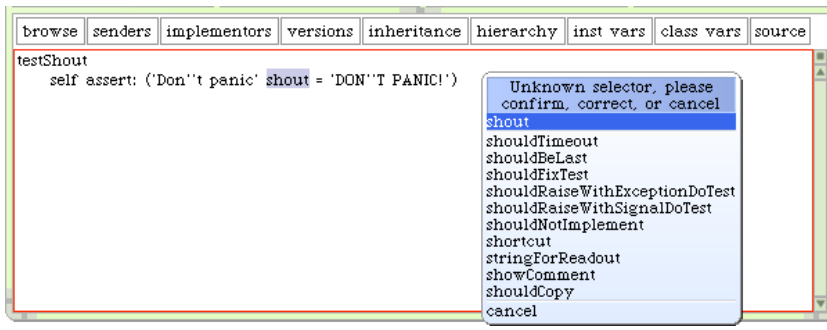


그림 1.21: StringTest 클래스의 testShout 메서드 수락

새로 만든 테스트를 실행하십시오. *Tools flap*에서 끌어 옮기거나 World ▷ open⋯ ▷ Test Runner 를 선택하여 SUnit TestRunner를 여십시오.

가장 왼쪽 두 개의 창은 시스템 브라우저의 상단 창과 약간 비슷합니다. 왼쪽 창에는 시스템 카테고리의 목록이 들어있지만, 이 창은 테스트 클래스를 포함하는 카테코리에 한정되어 있습니다.

Collections Tests-Text를 선택하면 오른쪽에 있는 패널은 클래스 StringTest를 포함한 카테고리의 모든 테스트 클래스를 보여줄 것입니다. 클래스 이름을 이미 선택했다면, 이들 테스트를 실행하기 위해 Run selected 를 클릭하십시오.

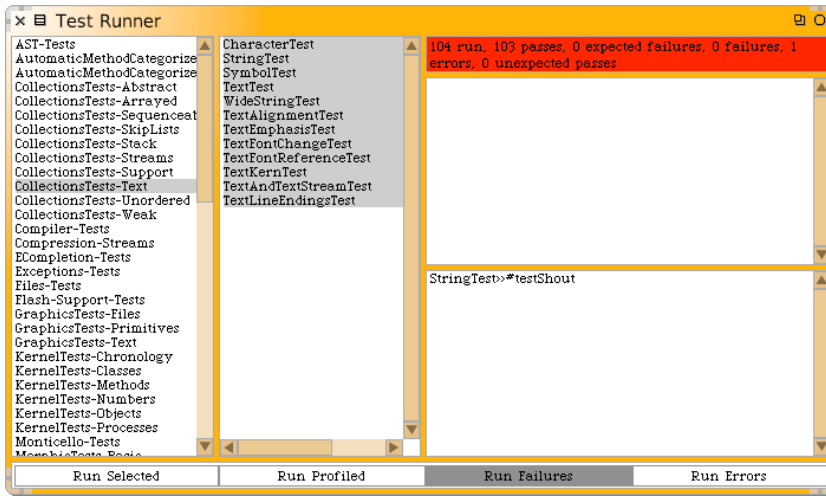


그림 1.22: String 테스트 실행

그림 1.22 에 보이는 것처럼 테스트 실행 도중 오류가 있었음을 알리는 메시지를 보게 됩니다. 오류를 일으키는 테스트의 목록은 하단 오른쪽 창에 보여줍니다. 보시다시피 `StringTest>>testShout`가 주범입니다. (참고로 `StringTest>>testShout`는 StringTest 클래스의 "testShout" 메서드를 식별하기 위한 스몰토크방식입니다.) 텍스트 줄을 클릭하면 오류가 생긴 테스트를 다시 실행할 것이며, 이 때 여러분은 "`MessageNotUnderstood:ByteString>>shout`"와 같은 식으로 발생한 오류를 보게 됩니다.

오류 메시지가 떠 있는 창이 바로 스몰토크디버거 입니다(그림 1.23 참조). 6장 에서 디버거와 이를 어떻게 사용하는지 보겠습니다.

물론 이 오류가 확실히 기대하던 바입니다. 테스트를 실행하면 오류가 발생하는데. 어떻게 "`shout`" 할 지 아직 작성하지 않았기 때문입니다. 그럼에도 불구하고, 테스트가 실패했음을 분명히 해두는 것은 바람직한 습관인데, 테스트 장치를 정확하게 설정했고 새 테스트가 실제로 실행중임을 확인시켜주기 때문입니다. 오류를 보고 나면, 디버거 창을 닫아서 실행 중인 테스트를 Abandon (포기) 할 수 있습니다. 참고로 스몰토크에서는 CreateNote 버튼을 사용하여 빠진 메서드를 정의하고, 새로 만든 메서드를 디버거에서 편집

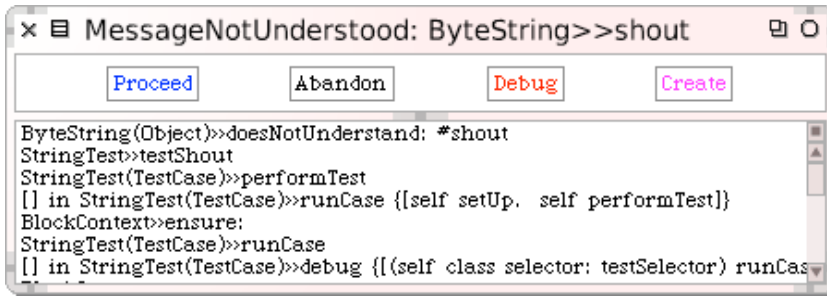


그림 1.23: (선) 디버거

하며, 테스트를 Proceed(진행)할 수 있습니다.

이제 테스트를 성공으로 이끌 메서드를 정의해보도록 하겠습니다!

시스템 브라우저에서 String 클래스를 선택하고, Converting(변환) 프로토콜을 선택한 후, 메서드 생성 템플릿에, 메서드 *1.2* 를 입력하시고 accept 를 실행하십시오(참고 : ^를 입력하려면, ^를 입력하십시오).

Method 1.2: "shout" 메서드

```
shout
    ↑ self asUppercase, '!'
```

쉼표는 문자열 결합 연산자입니다. 따라서 이 메서드 내용에서는 "String" 오브젝트와 "Shout" 메시지의 대문자 버전 문자열에 느낌표를 붙입니다. ^ 는 뒤에 이어진 표현식이 메서드에서 반환한 답이라는 것을 스퀵에 알려주며, 이 경우, 새롭게 연속된 문자열이 됩니다. 이 메서드가 동작할까요? 테스트를 실행하고 살펴보겠습니다.

테스트 러너에서 Run Selected 를 다시 클릭하면, 이번에는 모든 테스트가 실패와 오류없이 실행되었음을 가리키는 녹색 표시줄을 볼 수 있을 것입니다.

녹색 표시줄을 보셨다면,[5] 하시던 일을 저장하고 잠시 쉴 좋은 때입니다. 그러면 이제 좀 쉬도록 하겠습니다.

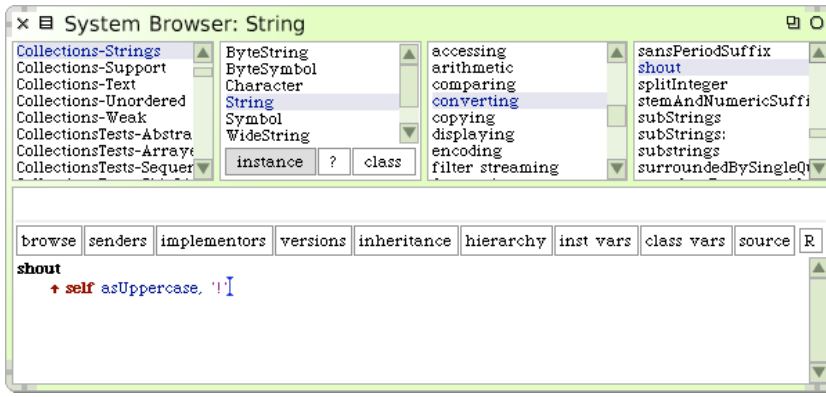


그림 1.24: 클래스 `String`에 정의된 `shout` 메서드

1.11 1장 요약

이 장에서는 스퀵환경을 소개하고, 시스템 브라우저(system browser), 메서드 파인더, 그리고 테스트 러너와 같은 몇 가지 주요 도구들의 사용법을 살펴보았습니다. 스퀵문법에 대해 모두 이해하진 못했지만 스퀵문법도 조금 보았습니다.

- 실행 중인 스퀵시스템은 가상 머신, sources 파일, image 파일, changes 파일로 구성되어 있습니다. 마지막 두 가지 요소만은 실행중인 시스템의 스냅샷을 기록할 때마다 바뀝니다.
- 스퀵image를 복구할 때, image에 가장 마지막으로 저장했을 때와 같은 객체들의 실행상태를 발견하게 됩니다.

[5] 실제로, 여러분은 몇가지 이미지가 버그 테스트를 포함하고 있고 여전히 수정될 필요가 있기 때문에 녹색 표시줄을 볼 수 없을 수도 있습니다. 이 문제에 대해 걱정하지 마십시오. 스퀵은 계속해서 진화하고 있습니다.

- 스퀵은 3-버튼 마우스로 다루도록 설계했습니다. 버튼은 빨강, 노랑, 파랑 버튼으로 알려져 있습니다. 3-버튼 마우스를 가지고 있지 않을 경우, 비슷한 동작을 하기 위해 수정자 키를 사용할 수 있습니다.
- 월드 메뉴를 불러오고 각종 도구들을 실행하기 위한 스퀵바탕 화면에서 빨강 버튼을 사용하십시오. 스퀵화면의 오른쪽에 있는 Tools flab 에서도 도구를 실행할 수 있습니다.
- Workspace는 코드를 작성하고 단편을 실행하기 위한 도구입니다. 임시 텍스트를 작성하기 위해 사용할 수도 있습니다
- Workspace 또는 다른 도구 내의 텍스트상에서 코드를 실행하기 위해 키보드 단축키를 사용할 수 있습니다. 이들 중 가장 중요한 것은 do it (cmd–d), print it (cmd–p), inspect it (cmd–i), explore it (cmd–I) 그리고 browse it (cmd–b) 입니다.
- SqueakMap 은 인터넷에서 쓸모 있는 패키지를 불러오기 위한 도구입니다.
- 시스템 브라우저는 스퀵코드를 검색하고, 새로운 코드를 개발하기 위한 주요 도구입니다.
- testRunner는 단위 테스트를 실행하는 도구입니다. 테스트 주도 개발을 지원하기도 합니다.

제 2 장

첫 번째 어플리케이션

이 장에서는 Quinto 라는 간단한 게임을 만들어보겠습니다. 게임을 만들면서 스퀵프로그래머들이 프로그램을 구현하고 디버그하는 작업에 사용하는 대부분의 도구들을 보여드리고, 다른 개발자들 프로그램을 바꾸는 과정을 보여드립니다. 시스템 브라우저, Object Inspector, 디버거, 그리고 몬티첼로*Monticello* 패키지 브라우저를 보여 드립니다. 스몰토크를 통한 개발 과정은 효율적입니다. 코드를 작성하는 작업에 훨씬 더 많은 시간을 보내지만, 훨씬 더 적은 개발과정을 거치게 됨을 발견할 것입니다. 프로그래밍 환경을 갖춘 도구가 언어와 잘 통합되어 있기도 하고, 스몰토크가 매우 단순하기 때문이기도 합니다.

2.1 Quinto 게임

스퀵프로그래밍 도구를 사용하는 방법을 보여드리기 위해, Quinto라는 단순한 게임을 만들겠습니다. 게임판은 그림 2.1로 보여드리며, 밝은 노란색 *칸*의 직사각형 배열로 구성되어 있습니다. 이 칸 들 중 하나를 클릭하면, 주위를 둘러싼 4개의 셀이 파랑색으로 바뀝니다. 다시 한 번 클릭하면 밝은 노란색으로 바뀝니다. 이 게임의 목표는 가능한 많은 색을 파랑색으로 바꾸는 것입니다.

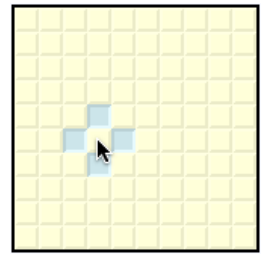


그림 2.1: Quinto 게임판입니다. 그림으로 보시는 바와 같이 사용자는 커서가 가리키는 위치에서 방금 클릭했습니다.

그림 2.1 에 보이는 Quinto 게임은 두 종류의 오브젝트로 제작했습니다. 오브젝트는 게임판과, 100개 각각의 칸 오브젝트입니다. 이 게임을 실행하는 스퀵코드는 두가지 클래스를 포함합니다. 하나는 게임에 대한 클래스고 다른 하나는 칸에 대한 클래스입니다. 스퀵프로그래밍 도구를 사용하여 클래스를 정의하는 방법을 보여 드리겠습니다.

2.2 새 클래스 카테고리 만들기

이미 1장 에서 시스템 브라우저를 보았고, 클래스와 메서드를 탐색하는 방법을 배웠으며, 새 메서드를 정의하는 방법을 보았습니다. 이제, 시스템 카테고리와 클래스를 만드는 방법을 보도록 하겠습니다.

 시스템 브라우저를 열고, 카테고리 창에서 노랑-클릭 하십시오. add item… 을 선택하십시오.

대화 상자에 새로운 카테고리 이름 (SBE-Quinto 를 사용할 것입니다) 을 입력하고, accept 을 클릭하시면 (또는 그냥 RETURN 키를 누릅니다), 새로운 카테고리를 만들고 카테고리 목록의 가장 마지막에 위치합니다. 기존의 카테고리를 먼저 선택했다면 선택한 부분의 바로 앞에 새 카테고리가 놓입니다.

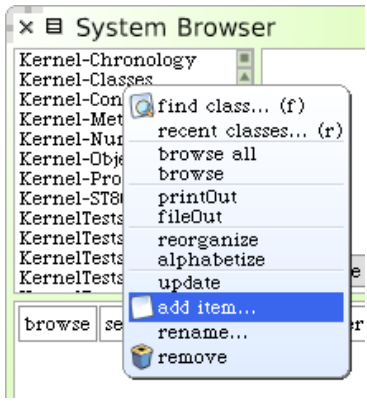


그림 2.2: 시스템 카테고리 추가

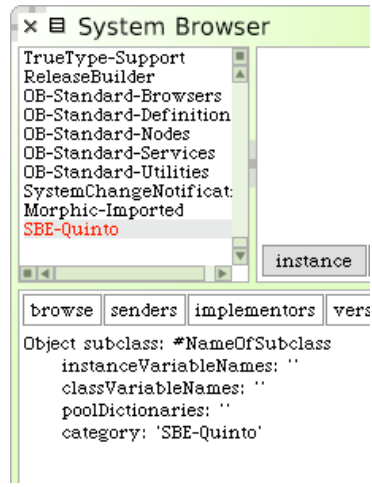


그림 2.3: 클래스 생성 템플릿

2.3 SBECell 클래스 정의하기

물론 새 카테고리에 아직 클래스가 없습니다. 하지만 주 편집 창에서는 새 클래스를 쉽게 만들 수 있도록 템플릿을 보여줍니다(그림 2.3을 보십시오).

이 템플릿은 Object 클래스에 NameofSubClass라고 하는 서브클래스를 만들어달라고 요청하는 메시지를 보내는 스몰토크구문을 보여줍니다. 새 클래스에는 변수가 없으며, 카테고리 *SBE-Quinto* 카테고리에 들어갑니다.

정말로 클래스를 만들어 보기 위해 템플릿을 간단히 수정해보겠습니다.

다음과 같이 클래스 생성 템플릿을 수정하십시오.

- `Object`를 `SimpleSwitchMorph`로 바꾸십시오.
- `#NameOfSubClass`를 `#SBECell`로 바꾸십시오.
- `mouseAction`을 인스턴스 변수 목록에 추가하십시오.

결과는 클래스 class 2.1 과 동일합니다.

Class 2.1: 클래스 SBECell 정의하기

```
SimpleSwitchMorph subclass: #SBECell
   instanceVariableNames: 'mouseAction'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'SBE--Quinto'
```

이 새 정의는 기존의 SimpleSwitchMorph 클래스에 SBECell 서브클래스를 만들어 달라고 요청하기 위해 메시지를 보내는 스몰토그 구문이 있습니다. (실제로, SBECell은 아직 없기 때문에, 앞으로 만들 클래스의 이름인 #SBEcell 심볼을 인자로 전달했습니다.) 또한 마우스를 칸 위에서 클릭할 때, 취해야 할 동작을 정의하려 사용하는 mouseAction 인스턴스 변수를 새 클래스의 인스턴스가 갖도록 언급하였습니다.

이 시점에서 여전히 아무 것도 만들지 않았습니다. 참고로 클래스 템플릿 창의 테두리가가 빨강색으로 바뀌었습니다(그림 2.4). 저장하지 않은 바뀐 내용이 있음을 의미합니다. 이 메시지를 실제로 보내려면 accept 해야 합니다.

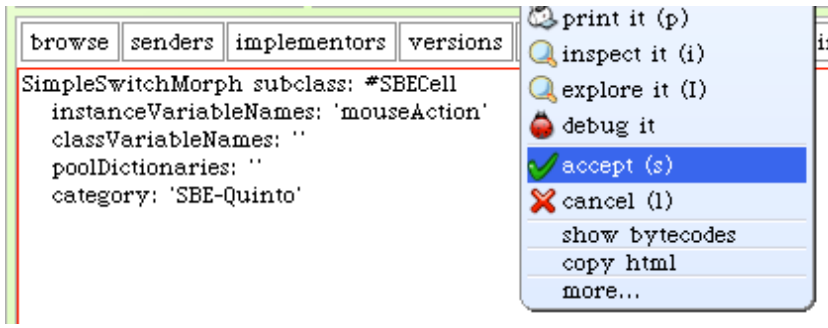


그림 2.4: 클래스 생성 템플릿(The class-creation Template)

새 클래스 정의를 *accept*하십시오.

노랑-클릭을 하고 accept를 선택하거나, (저장을 위해) CMD-S 단축키를 사용하십시오. 새 클래스를 컴파일 하도록 SimpleSwitchMorph에 메시지를 보낼 것입니다.

클래스 정의를 accept 하면 클래스를 만들고 브라우저의 클래스 창에 나타납니다(그림 2.5). 편집 창에서는 이제 클래스 정의를 보여주면, 그 아래의 작은 창에서 클래스의 목적에 대한 설명을 간단하게 작성하도록 알릴 것입니다. 이를 클래스 주석이라고 하며, 이 클래스의 고급 개요를 다른 프로그래머들에게 제공하기 위해 작성하는 상당히 중요한 역할을 담당합니다. 스몰토커는 코드의 가독성에 대한 가치를 매우 높게 쳐주며, 메서드에 자세한 주석을 달 필요는 없습니다: 코드의 내용은 코드 자신이 만들어야 한다는 철학입니다 (만약 그렇지 못하면, 코드가 자신을 제대로 설명할 때까지 셰속 리팩토링해야 합니다!). 클래스 주석은 그 클래스에 대한 자세한 설명을 넣을 필요는 없지만, 당신의 뒤에 따라오게될 프로그래머가 이 클래스를 살펴보는데 시간을 소모할 것 같으면, 전체적인 목적에 대한 간단한 설명은 단연 필수입니다.

 `SBEcell`에 대한 클래스 주석을 입력하고, *accept it*하시면 다음에도 계속 개선할 수 있습니다.

2.4 클래스에 메서드 추가하기

이제 클래스에 몇 가지 메서드를 추가해보도록 하겠습니다.

 프로토콜 패널에서 --all-- 프로토콜을 선택하십시오.

편집 창에서 메서드 생성 템플릿을 보게 될 것입니다. 이를 선택하고 메서드 2.2 의 내용으로 바꾸십시오.

그림 2.5: 새롭게 만든 클래스 SBECell

Method 2.2: SBECell의 인스턴스 초기화

```
initialize
   super initialize.
   self label: ''.
   self borderWidth: 2.
   bounds := 0@0 corner: 16@16.
   offColor := Color paleYellow.
   onColor := Color paleBlue darker.
   self useSquareCorners.
   self turnOff
```

참고로 3번째 줄에 있는 '' 문자는 큰 따옴표가 아니라 사이에 아무것도 없는 두 개의 분리된 작은 따옴표입니다! '' 는 빈 문자열을 나타냅니다.

 이 메서드 정의를 Accept 하십시오.

이 코드가 하는 역할이 무엇일까요? 여기서는 모든 자세한 내용을 다루지는 않겠지만(이 책의 나머지 부분에서 다룰 내용입니다!), 간단한 미리 보기를 보여드리겠습니다. 하나하나 살펴보겠습니다.

"initialize"라고 하는 메서드를 보겠습니다. 이름은 굉장히 중요합니다! 관례에 따라, 클래스에서 "initialize"라는 메서드를 정의하면, 오브젝트를 만들고 나서 바로 호출됩니다. 따라서, "SBECell new"를 실행하면, "initialize"메시지를 새로 만든 오브젝트에 자동으로 보냅니다. initialize 메서드는 오브젝트의 상태를 설정하는데 사용하며, 보통 인스턴스 변수를 설정합니다. 이것이 정확히 여기서 할 일입니다.

이 메서드가 수행하는 첫 번째 동작(두번째 줄)은 "SimpleSwitchMorph" super클래스의 initialize 메서드를 실행하는 것입니다. 이 개념은 어떤 상속받은 상태든지 상위 메서드의 initialize 메서드를 통해 올바르게 초기화된다는 것입니다. 어떤 다른 동작을 수행하기 전에, super initialize를 보내어 상속받은 상태를 초기화하는 것은 언제나 바람직한 생각입니다. SimpleSwitchMorph의 initialize 메서드가 무얼 할지에 대해서는 정확히 모르며 상관하지도 않지만, 타당한 기본 값을 유지하는 일부 인스턴스 변수를 설정하는데 있어서는 최선의 방법이므로, 이를 호출하는 것이 더 나으며, 그렇지 않으면 깔끔하지 않은 상태로 시작할 위험에 빠져들게 됩니다.

이 메서드의 나머지 부분에서는 이 객체의 상태를 설정합니다. *self label:' '* 을 보내면, 이 객체의 레이블을 빈 문자열로 설정gkq니다.

표현식 0@0 corner:16@16 은 아마도, 약간의 설명이 필요할거같네요. 0@0은 x와 y 좌표 각각을 0으로 설정하는 *Point* 객체입니다. 0@0은 메시지 @를 인자 0과 함께 숫자 0으로 보냅니다. 이에 대한 결과로, 숫자 0이 Point 클래스에 좌표 (0,0)에 대한 새 인스턴스를 만들어 달라고 요청할 것입니다. 이제 모서리 0@0과 16@16으로 Rectangle을 만들도록 하는 새롭게 만든 점 메시지 corner:16@16을 보냅니다. 새롭게 만든 직사각형은 super클래스에서 상속된 bounds에게 할당할 것입니다.

스퀵화면의 기준점은 상단 왼쪽이며, y 좌표는 아래 방향으로 된다는것에 유의하십시오.

메서드의 나머지 부분은 자체적으로 설명할 수 있어야 합니다. 좋은 스몰토크코드를 작성하는 예술적인 측면은 좋은 메서드 이름을 골라서 스몰토크코드가 쉬운 영어(pidgin English)처럼 보일 수 있게 하는 것입니다. 오브젝

트가 자신과 대화하면서 "Self use square corners!", "Self turn off" 라고 말하는 것을 상상할 수 있어야 합니다.

2.5 객체 점검하기

이제부터 새로운 `SBCcell` 객체를 만들고 그 객체를 점검함으로써, 작성한 코드의 효과를 테스트할 수 있습니다.

워크스페이스를 여십시오. `SBECell new` 구문을 입력하고 inspect it 을 실행하십시오.

Inspector 의 왼쪽 창은 인스턴스 변수의 목록을 보여줍니다. 여러분이 하나의 변수를 선택하면 (`bounds`를 선택해보십시오), 인스턴스 변수 의 값을 오른쪽 창에 보여줍니다. 인스턴스 변수의 값을 바꾸기기 위해 Inspector 를 사용할 수 있습니다.

`bounds`의 값을 `0@0 corner:50@50` 으로 바꾸고 accept 하십시오.

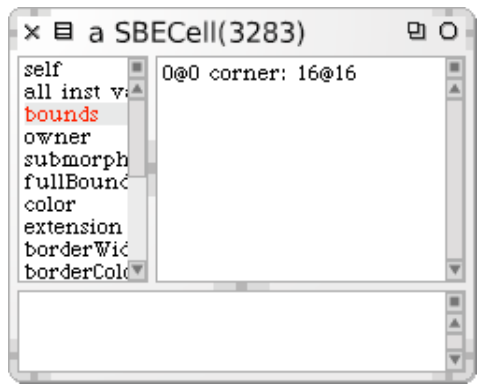


그림 2.6: SBECell 객체를 검사하기 위해 사용된 Inspector

Inspector의 하단 창은 작은 워크스페이스입니다. 이 워크스페이스는 가상 변수 self 가 점검중인 객체에 묶이므로 쓸모가 있습니다.

하단 창에서 `Self openInWorld`를 입력하고, do it 을 실행하십시오.

칸은 bounds가 나타나야 한다고 언급하는 위치에 정확하게 화면 상단 왼쪽 구석에 반드시 나타나야 합니다. 모픽 할로를 불러오기 위해 셀에서 파랑-클릭을 하십시오. 갈색 손잡이(상단 오른쪽 옆)로 칸을 옮기고, (하단-오른쪽의) 노랑 손잡이로 크기 조절 하십시오. Inspector가 보고한 bound도 어떻게 바뀌는지 보십시오.

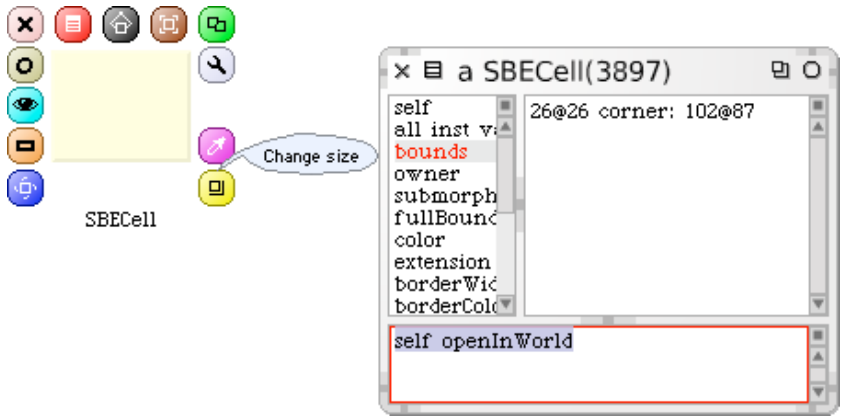


그림 2.7: 칸 크기 조절.

분홍 손잡이의 x를 눌러 칸을 지우십시오.

2.6 SBEGame 클래스 정의하기

이제 SBEGame이라고 할 게임에 필요한 다른 클래스를 만들어보도록 하겠습니다.

브라우저 메인 창에서 클래스 정의 템플릿을 보이게 하십시오.

이미 선택한 클래스 카테고리의 이름을 두 번 클릭하거나, (instance 버튼을 클릭하여) SBECell의 정의를 보이게 하십시오. 다음과 같이 코드를 편집하고 accept 하십시오.

Class 2.3: SBEGame 클래스 정의

```
BorderedMorph subclass: #SBEGame
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'SBE--Quinto'
```

이제 BoderedMorph의 서브클래스를 만들 차례입니다.[1] Morph는 스퀵의 모든 그래픽 도형에 대한 super클래스이며 (놀랍군요!), BorderedMorph는 테두리를 가진 Morph입니다. 두번째 줄의 따옴표 사이에 인스턴스 변수의 이름을 넣을 수 있겠지만, 지금은 그냥 빈 목록으로 남겨 놓겠습니다.

자 이제 SBEGame에 대한 initialize 메서드를 정의하도록 하겠습니다.

 SBEGame에 대한 메서드를 브라우저에 다음과 같이 입력하고, accept 해 보십시오:

Method 2.4: 게임 초기화

```
1  initialize
2     | sampleCell width height n |
3     super initialize.
4     n := self cellsPerSide.
5     sampleCell := SBECell new.
6     width := sampleCell width.
7     height := sampleCell height.
8     self bounds: (5@5 extent: ((width*n) @(height*n)) + (2 * self
       borderWidth)).
9     cells := Matrix new: n tabulate: [ :i :j | self newCellAt: i
       at: j ].
```

스퀵은 용어 몇 가지의 이름을 모른다고 불평할 것입니다. 스퀵은 CellsPerSide 메시를 모른다고 할 것이며, 수많은 컬렉션을 제안할텐데, 이 경우에는 철자 오류가 있습니다.

하지만 cellsPerSide는 잘못 입력한 것이 아닙니다. — 아직 정의하지 않은 메서드일 뿐입니다 — 잠깐동안 메서드를 정의해보겠습니다.

[1] Package가 이럴 때 유용합니다. 여러 개의 클래스를 하나로 묶는거죠. 즉 super클래스가 서로 다른 여러개의 클래스를 같은 패키지로 묶어서 라이브러리처럼 관리합니다

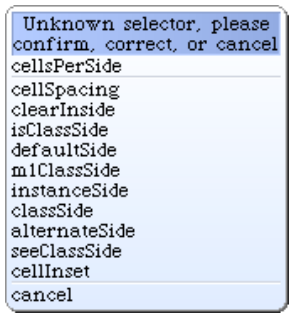


그림 2.8: 알려지지 않은 selector를 감지한 스퀵

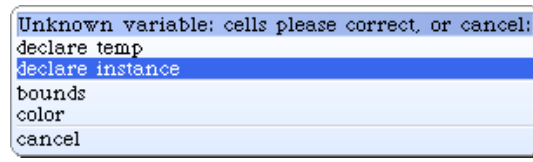


그림 2.9: 새로운 인스턴스 변수 선언하기

따라서 정말로 의미했던 `cellsPerside`를 확인하기 위해 메뉴에서 첫번째 항목을 선택하십시오.

다음, 스퀵은 cells의 의미를 모른다고 불평할 것입니다. 이런 문제를 수정하기 위해 수많은 방법을 제공합니다.

*cells*를 인스턴스 변수로 만들 것이므로, declare instance를 선택하십시오.

마지막으로, 스퀵은 마지막 줄에서 보낸 newCellAt:at: 메시지에 대해 불평할 것입니다. 이것 역시 실수가 아니므로 메시지 확인합니다.

클래스 정의를 다시 한 번 보았다면 (instance 버튼을 클릭하면 실행할 수 있음), 인스턴스 변수 cells를 포함하기 위해 브라우저가 변수의 정의를 적용[2] 했다는 것을 알게 될 것입니다.

이 initialize 메서드를 보겠습니다. `| sampleCell width height n |` 줄에서 임시로 4개의 변수를 선언합니다. 변수의 범위와 수명이 메서드에 국한되므로 임시 변수라고 부릅니다. 설명을 잘하는 이름이 붙은 임시 변수는 코드 가독성을 더 좋게 하는데 도움이 됩니다. 스몰토크에는 상수와 변수를 구분하기

[2] Class가 정의된 부분을 보면 instanceValirableNames 부분에 cells가 추가된것을 확인할 수 있습니다

위한 특별한 문법이 없으며, 사실 이 네 개의 모든 "변수"는 사실 상수입니다. 4-7번째 줄은 이들 상수를 정의합니다.

게임판은 얼마나 커야 할까요? 몇 가지 정수 만큼의 칸을 담기에 충분히 크고, 그 칸 주위에 경계를 그리기에 충분한 커야 합니다. 칸 갯수는 얼마가 적당할까요? 5? 10? 100? 아직은 모르지만, 이미 알고 있었다면 나중에 바꿀지도 모릅니다. 그러므로, cellsPerSide라고 하는 다른 메서드로 숫자를 알려줄 책임을 부여하고 잠시동안 이대로 작성하도록 하겠습니다. initialize에 대한 메서드 내용을 accept 했을 때, "확인(confirm), 수정(correct) 또는 취소(cancel)" 를 물어보는 다이얼로그가 뜨게되는데 이것은 스퀵이 요청한 이름을 가진 메서드가 정의되기 이전에, cellsPerSide라는메시지를 보내기 때문입니다. 걱정하지 마십시오. 사실 이런 방법은 아직 정의하지 않은 다른 메서드를 가지고 작업을 진행하는 좋은 방법입니다. 왜일까요? 글쎄요, 우리가 이것이 필요하다고 깨달은 시점은 initialize 작성을 시작하기 전의 상황은 아니며, 지금 알았다고 해도 현재의 작업의 흐름을 방해하지 않으면서 진행중에 의미있는 이름을 부여하고 옮길 수 있기 때문입니다.

네번째 줄은 이렇게 정의된 cellsPerSide 메서드를 사용합니다. 스몰토크에서 self cellsPerSide는 cellsPerSide 메시지를 나 자신(SBEGame)을 의미하는 객체인 self에 보냅니다. 게임판의 각 모서리당 칸의 갯수 응답을 n에 할당합니다.

다음 세 줄에서는 새로운 SBECell 오브젝트를 만들고, 오브젝트의 너비와 높이를 적당한 임시 변수에 넣습니다.

여덟번째 줄은 새로운 오브젝트의 bounds를 설정합니다. 지금까지의 세부적인 내용에 대한것은 걱정하지 마시고, 괄호안의 표현식은 점 (5,5)에서 원점(예를 들어, 상단 왼쪽 구석)으로 사각형을 만들고, 하단 오른쪽 모서리는 셀의 오른쪽 숫자만큼의 공간을 허용하기 위해 충분히 떨어져 있을 것이라는 설명을 믿어주세요.

마지막 줄은 `SBEGame` 오브젝트의 인스턴스 변수 cells에 올바른 행 렬 갯수대로 새로 만든 *Matrix*를 할당합니다. Matrix 클래스(클래스도 오브젝트이므로, 메시지를 보낼 수 있습니다)에 `new:tabulate:`메시지를 보내어 처리했

습니다. new:tabulate:의 이름에 두 개의 콜론(:)이 있는걸 보고, 우리는 이것이 인자를 두 개 받는다는걸 알 수 있습니다. 인자는 콜론 바로 다음에 위치합니다. 만약 당신이 인자를 모두 괄호 안에 넣는 프로그램 언어를 사용해왔었다면, 이 표기법이 매우 이상하게 느껴질 것입니다. 당황하지 마세요, 이것은 단지 문법일 뿐입니다! 이런방법은 메서드의 이름으로 인자의 역할을 설명하는데 활용할 수 있어 상당히 좋은 문법으로 알려져 있습니다. 예를 들어, `Matrix rows: 5 columns: 2` 는 5개의 가로 열과 2개의 세로 열을 갖는 것이지 2개의 가로 열과 5개의 세로 행을 가진 것이 아니라는 점을 확실히 알게 합나다.

`Matrix n tabulate: [ :i :j | self newCellAt: i at: j ]` 은 새로운 n x n 2차원 배열을 만들고 요소를 초기화합니다. 각각의 초기 값은 좌표에 따라 다릅니다. (i,j)번째 요소는 newCellAt: i at: j. 처리 결과로 초기화합니다.

이것이 바로 `initialize` 입니다. 이 메시지 내용을 accept 하면, 아마도 작성한 코드를 가지런히 다듬고(pretty-up formating) 싶을 것입니다. 직접 일일히 할 필요는 없고, 노랑-버튼 메뉴에서, more ▷ prettyprint 를 선택하면, 사용자를 위해 브라우저가 알아서 해줄 것입니다. 메서드를 깔끔하게 출력하고 난 후에, 다시 accept 하거나, 결과가 맘에 들지 않으면 물론 취소 (CMD-l 숫자 1이 아닌 L의 소문자입니다)할 수 있습니다. 대신, 코드가 보일때는 언제나 pretty-printer 를 브라우저가 사용하도록 설정할 수 있습니다. 뷰를 조절하려면 버튼 표시줄에서 가장 오른쪽 버튼을 사용하십시오.

more⋯ 를 더 많이 사용하신다면, more⋯ 를 직접 불러오려고 클릭할 때 Shift 키를 누른 채로 유지한다는 것을 알고 계시는 것이 좋습니다.

2.7 프로토콜에 메서드 연계하기

좀 더 다양한 메서드를 정의하기 전에, 브라우저의 상단에 있는 세번째 패널을 잠시 보겠습니다. 브라우저의 첫 번째 패널은, 클래스를 카테고리화해서, 두 번째 패널에 있는 매우 긴 클래스 목록에 사용자가 놀라지 않게 해주며, 동일한 방식으로 세 번째 패널은 네 번째 패널에 있는 매우 긴 메서드 목록

에 당황하지 않게 해줍니다. 이러한 메서드의 카테고리를 "프로토콜" 이라고 부릅니다.

클래스에 단지 몇 개의 메서드만 존재한다면, 프로토콜이 제공하는 계층형 추가 레벨은 정말 필요하지 않습니다. 브라우저에서 클래스의 모든 메서드를 포함하는 --all-- 이라고 되어있는 가상 프로토콜을 제공하므로 우리가 배울 때 놀랄 필요는 없습니다.

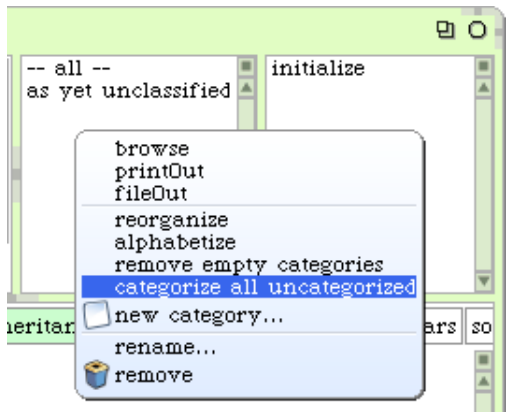


그림 2.10: 분류하지 않은 모든 메서드의 분류.

당신이 이 예제를 따라 하셨다면, 세 번째 패널은 *아직 분류되지 않은* 프로토콜을 포함할 가능성이 높습니다.

노랑-버튼 메뉴 항목을 선택하고 이 항목을 수정하기 위해 `categorize all uncategorized`를 선택한 후, initialization 이라고 하는 새로운 프로토콜에 `initialize` 메서드를 이동하십시오.

어떻게 이게 올바른 프로토콜인지 스퀵이 알았을까요? 글쎄요, 일반적인 경우 스퀵은 알지 못하지만, 이 경우에서는 super 클래스에 `initialize` 메서드가 있기 때문에, 스퀵에서 `initialize` 메서드를 재지정한 같은 카테고리로 이동해야 겠다고 가정했습니다.

스퀵이 이미 `initialize` 메서드를 initialization 프로토콜에 넣었음을 발견했습니다. 정말 그렇다면, 아마 `AutomaticMethodCategorizer`라는 패키지를

사용자의 이미지에 불러왔기 때문일지도 모릅니다.

표기 관례. 스몰토커는 메서드가 속하는 클래스를 구분하기 위해 ">>" 표기를 자주 사용하므로, `SBEGame`에 있는 `cellsPerSide` 메서드는 `SBEGame>>cellsPerSide`와 같이 참조합니다. 이 부분이 스몰토크문법이 아님을 나타내기 위해, 우리는 대신 `>>` 라는 특별한 심볼을 사용하여, 본문상에서 `SBEGame>>cellsPerSide`처럼 보이도록 표현하겠습니다.

지금부터 이 책에서 메서드를 보여드릴 때는, 이런 양식으로 메서드의 이름을 작성할 것입니다 물론, 브라우저에 코드를 실제로 타이핑 할 때, 클래스 이름 또는 `>>` 기호를 입력할 필요가 없으며, 단지 클래스 패널에 적당한 클래스를 선택했는지만 확인하면 됩니다.

자 이제, `SBEGame>>initialize` 메서드에서 사용하는 다른 두 개의 메서드를 정의하도록 하겠습니다. 둘 모두 `initialization` 프로토콜로 이동할 수 있습니다.

Method 2.5: 상수 메서드

```
SBEGame>>cellsPerSide
   "The number of cells along each side of the game"
   ↑ 10
```

이 메서드는 더 이상 단순해 질 수 없습니다. 이 메서드는 상수 10을 반환합니다. 메서드로서 상수를 나타내 장점은, 프로그램이 발전해서 상수가 몇 가지 다른 특성에 의존하는 경우, 메서드는 계산등을 위해 현재 값 10을 변경할 수 있습니다.

Method 2.6: 헬퍼 메서드 초기화

```
SBEGame>>newCellAt: i at: j
   "Create a cell for position (i,j) and add it to my on--screen
   representation at the appropriate screen position.  Answer the
    new cell"
   | c origin |
   c := SBECell new.
   origin := self innerBounds origin.
   self addMorph: c.
   c position: ((i -- 1) * c width) @ ((j -- 1) * c height) +
    origin.
   c mouseAction: [self toggleNeighboursOfCellAt: i at: j].
```

 메서드 `SBEGame>>cellsPerSide` 와 `SBEGame>>newCellAt:at:`를 추가하십시오.

새로운 셀렉터 `toggleNeighboursOfCellAt:at:`과 `mouseAction:`의 철자를 확인하십시오.

메서드 2.6 은 게임상의 칸에 대한 Matrix에서 특정 위치 (i,j) 에 특별히 지정된 새로운 `SBECell`로 응답합니다. 마지막 라인은 블록 `[self toggleNeighboursOfCellAt: i at: j ]`이 되도록 새로운 칸의 `mouseAction`을 정의합니다. 실제로, 이 정의는 마우스를 클릭할 때 실행할 `callback` 동작을 정의합니다. 이에 대응하는 메서드 또한 정의해야할 필요가 있습니다.

Method 2.7: *Callback* 메서드

```
SBEGame>>toggleNeighboursOfCellAt: i at: j
   (i > 1) ifTrue: [ (cells at: i -- 1 at: j ) toggleState].
   (i < self cellsPerSide) ifTrue: [ (cells at: i + 1 at: j)
    toggleState].
   (j > 1) ifTrue: [ (cells at: i  at: j -- 1) toggleState].
   (j < self cellsPerSide) ifTrue: [ (cells at: i at: j + 1)
    toggleState].
```

메서드 2.7 은 게임상 (i,j) 칸의 사방에 있는 4개의 칸 상태를 토글합니다. 유일한 문제가 있다면 보드는 유한하므로, 우리가 칸의 상태를 토글 하기 전에, 이웃한 칸이 존재하도록 철저히 대비를 해두어야 한다는 것입니다.

 `game logic`이라 불리는 새 프로토콜에 이 메서드를 배치하십시오.

메서드를 이동하려면, 간단하게 이름을 클릭하고, 새로 만든 프로토콜(그림 2.11)로 끌어다 놓습니다.

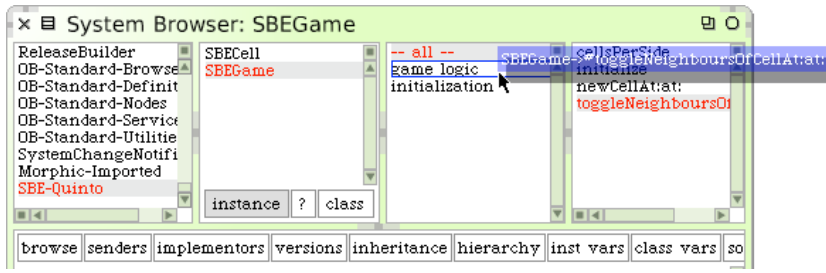


그림 2.11: 메서드를 프로토콜에 드래그하기

Quinto 게임을 완성하려면, 마우스 이벤트를 다루기 위해 SBEcell 클래스에서 두 개 이상의 메서드를 정의할 필요가 있습니다.

Method 2.8: 전형적인 *setter* 메서드

```
SBECell>>mouseAction: aBlock
   ↑ mouseAction := aBlock
```

메서드 2.8 은 인수로서 칸에 대한 mouseAction 변수를 설정하고 새로운 값으로 답하는 것 이외에 다른 어떤 역할도 하지 않습니다. 이러한 방식으로 인스턴스 변수의 값을 바꾸는 메서드를 *setter* 메서드라고 하며, 인스턴스 변수의 현재 값을 답하는 메서드를 *getter* 메서드라 부릅니다.

당신이 다른 프로그래밍 언어로 getters와 setters를 사용하셨다면, `setmouseAction`과 `getmouseAction`이라고 하는 메서드를 기대했겠지만, 스몰토크의 표기 관례는 다릅니다. getter는 변수가 갖는 이름과 동일한 이름으로 지어져야 하며, setter는 유사한 이름으로 만들어지지만, 마지막에 ":"를 붙입니다. 따라서 `mouseAction`과 `'mouseAction:'`입니다.

정리하자면, setter와 getter는 접근자 메서드라고 불리며, 관례상, accessing 프로토콜에 있습니다. 스몰토크에서, 모든 인스턴스 변수는 오브젝트에

서 private으로 취급되며, 스몰토크언어에서 다른 오브젝트를 통해 이들 인스턴스 변수를 읽고 쓰려면 이와 같은 접근자 메서드[3] 를 통해야 합니다.[4]

SBECell 클래스로 가서, SBECell>>mouseAction:을 정의하고 accessing 프로토콜에 넣으십시오.

마지막으로 메서드 mouseUp:을 정의해야 하며, 이 메서드는 화면의 칸 위에 마우스가 머무는 동안 마우스 버튼을 떼면, GUI 프레임워크에서 자동으로 호출할 것입니다.

Method 2.9: 이벤트 핸들러

```
SBECell>>mouseUp: anEvent
   mouseAction value
```

SBECell>>mouseUp: 메서드를 추가하고, 메서드를 categorize all uncategorized 하십시오.

이 메서드는 인스턴스 변수 mouseAction에 저장한 객체로 value 메시지를 보내서 처리하도록 동작합니다. 사용자가 SBEGame>>newCellAt: i at: 에서 mouseAction에 인자로 대입한 다음의 코드가 다시 호출됩니다:

```
[self toggleNeighboursOfCellAt: i at: j ]
```

value 메시지를 보내면, 이 코드부분이 실행되며 칸의 상태를 전환합니다.

2.8 코드를 실행해봅시다

다 되었습니다. Quinto 게임을 완성했습니다!

모든 단계를 따라왔다면, 2개의 클래스와 7개의 메서드만으로도 게임을 즐길 수 있습니다.

[3] 사실은 서브클래스에서는 접근이 가능합니다

[4] 이것은 객체의 속성중 캡슐화에 관련된 부분입니다. http://ta.onionmixer.net/wordpress/?p=126이 내용을 참고해주세요

 워크스페이스에서 `SBEGame new openInWorld`를 입력하고 `do it`을 실행하십시오.

게임이 열리고 나면, 각각의 칸을 클릭해서 어떻게 동작하는지 보실 수 있습니다.

여튼, 잡담이 길었군요 칸을 클릭하면, PreDebugWindow창이라고 하는 알림 창에서 *오류 메시지*를 나타납니다!

아래의 그림 2.12 에서 나타난 바와 같이, `MessageNotUnderstood: SBEGame >>toggleState`라고 하고 있습니다.

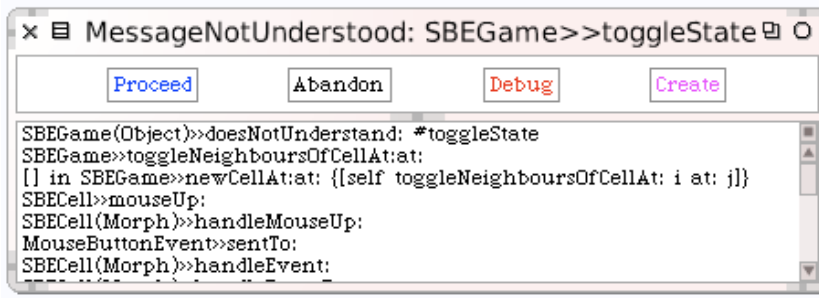


그림 2.12: 칸을 클릭했더니 게임에 버그가 있네요!

어떤 일이 일어났을까요? 그것을 알아내기 위해, 좀 더 강력한 스몰토크의 도구인 디버거 *Debugger* 를 사용해 봅시다.

 알림 창에서 `debug` 버튼을 클릭하십시오.

디버거가 나타날 것입니다. 디버거 창의 윗 부분에서 활성화 된 모든 메서드를 보여주는 실행 스택을 볼 수 있을 것이며, 스몰토크코드는, 강조한 오류를 트리거한 부분과 함께 선택한 메서드를 가운데 창에서 실행하는 중입니다.

 *(상단 부근의)* `SBEGame>>toggleNeighboursOfCellAt:at:`이라는 레이블이 붙은 줄을 클릭하십시오.

디버거는 오류가 발생한 이 메서드 안의 실행 상태를 보여줄 것입니다(그림 2.13).

디버거의 하단에는 두 개의 작은 Inspector 창이 있습니다. 왼쪽에는 커서로 현재 선택된 메서드를 실행하기위해 요청한 메시지(메서드)의 수신자 오브젝트를 확인해서 해당 시점의 인스턴스 값을 볼 수 있습니다. 오른쪽에서는 현재 실행 중인 메서드 자체를 나타내는 오브젝트를 확인하여 메서드의 매개변수와 임시 변수 값을 볼 수 있습니다.

그림 2.13: 선택된 메서드 `toggleNeighboursOfCell:at:`와 함께 있는 모습의 디버거

디버거를 사용하면, 단계적으로 코드를 실행하고 매개 변수와 로컬 변수에서 오브젝트를 점검하고, 워크스페이스에서 여러분이 할 수 있는 작업과 동등하게 코드를 시험할 수 있습니다. 그리고 가장 놀라운 것은, 다른 디버거들에 사용된 코드들도 시험할 수 있고, 디버그 진행 중에 코드를 바꿀 수도 있다는 것입니다. 몇몇 스몰토크사용자들은 브라우저에서 보다는 대부분의 경우 디버거에서 프로그램을 작성합니다. 디버거에서 프로그램을 작성하는 장점은, 실제 실행 상황에서 실제 매개 변수와 함께, 여러분이 작성하고 있는 앞으로 실행할 메서드를 살펴볼 수 있다는 것에 있습니다.

이 경우, SBEGame의 인스턴스에 완전히 머물러 있는 동안 SBECell의 인스턴스로 보낸 toggleState 메시지를 상단 패널의 첫번째 줄에서 볼 수 있습

니다. 원인은 추측하건데 cells 2차원 배열의 초기화에 있는 것 같습니다. `SBEGame>>initialize` 코드를 보면 newCellAt:at:의 반환 값으로 cells를 채워 놓음을 보여주지만, 우리가 해당되는 메서드를 살펴보니, 메서드에 반환문이 없습니다! `newCellAt:at:`의 경우 메소드는 기본적으로 self 인스턴스로 값을 리턴합니다. newCellAt:at:의 경우에는 확실히 SBEGame의 인스턴스이기 때문에 (일단 SBEGame의 셀렉터이기도 하고) 제대로된 값을 반환하기 위해서는 SBECell 인스턴스를 값으로 반환해야 합니다.

 디버거 창을 닫으십시오. `SBEGame>>newCellAt:at:` 메서드 마지막 부분에 *"^ c"* 구문을 추가하여, `c` 를 반환하게 하십시오 *(메서드 2.10을 봐주세요).*

Method 2.10: 버그 수정

```
SBEGame>>newCellAt: i at: j
   "Create a cell for position (i,j) and add it to my on--screen
   representation at the appropriate screen position.  Answer the
    new cell"
   | c origin |
   c := SBECell new.
   origin := self innerBounds origin.
   self addMorph: c.
   c position: ((i -- 1) * c width) @ ((j -- 1) * c height) +
    origin.
   c mouseAction: [self toggleNeighboursOfCellAt: i at: j].
   ↑ c
```

1장 에서 보았듯이, 스몰토크에서 메서드내부에서 값을 반환하기 위해 사용하는 기호는 ^를 입력함으로써 얻는 ^ 입니다.

종종 디버거 창에서 코드를 직접 수정할 수 있으며, 프로그램 실행을 계속하려면 Proceed를 클릭합니다. 이번의 경우, 버그는 오류가 발생한 메서드가 아닌, 오브젝트의 초기화 과정에서 생겼으므로, 가장 쉽게 코드를 수정하는 방법은 디버거 창을 닫고, 실행 중인 게임의 인스턴스를 (할로와 함께) 폐기한 후, 새 인스턴스를 만드는 것입니다.

 *Do:* `SBEGame new openInWorld` 를 다시 실행하십시오.

이제 게임이 제대로 동작할 것입니다.

2.9 스몰 토크 코드를 저장하고 공유하기

이제 Quinto 게임이 동작하면, 친구와 공유할 수 있도록 어딘가에 저장해야 할 것입니다. 물론 여러분은 자신의 전체 스퀵이미지를 저장할 수 있으며, 이 이미지를 실행하여 여러분의 첫 번째 프로그램을 자랑할 수 있지만, 여러분의 친구들은 아마도 그들의 이미지에, 자신들만의 코드를 포함하려 할 것이고, 여러분의 이미지를 사용하기 위해 그 코드를 포기하고 싶지는 않을겁니다. 필요한 것은 여러분의 스퀵이미지 외부에서 소스 코드를 얻는 방법이며, 이대로 하면 다른 프로그래머들이 그 소스코드를 자신의 것으로 만들 수 있을 것입니다.

이 작업을 수행하는 가장 쉬운 방법은 코드를 파일로 내보내기하는 것입니다. 시스템 카테고리 창에서 노랑 버튼 메뉴는, SBE-Quinto 카테고리의 전체 내용을 파일로 내보내기 위한 옵션을 제공할 것입니다. 이런 작업의 결과로서 나온 파일은 사람이 읽기에는 좀 더 어렵지만, 컴퓨터용이지 사람을 위한 것은 아닙니다. 이 파일을 친구에게 이메일로 보낼 수 있으며, 친구들은 파일 목록 브라우저*the file list browser* 를 사용하여, 그들의 스퀵이미지에 파일을 넣을 수 있습니다.

 `SBE-Quinto` 카테고리에서 노랑 클릭하고 내용을 `fileOut` 하십시오.

여러분의 이미지를 저장한 디스크의 같은 폴더에서 "SBE-Quinto.st" 라고 하는 파일을 찾을 수 있습니다. 텍스트 편집기로 파일을 살펴보도록 하겠습니다.[5]

 최신 스퀵이미지를 열고 *SBE-Quinto.st* 파일 출력을 file in 하기 위해 파일 목록 도구*(File List tool)* 를 사용하십시오. 새 이미지에서 게임의 동작을 검증하십시오.

[5] one click버전은 Contents/Resources/ 디렉토리 안에 파일이 있습니다. 클래스이름의 대소문자까지 정확하게 구분하니 참고하는것이 좋고 저 Resources 디렉토리는 squeak의 image가 있는 디렉토리니 다른경로에 image를 두는경우도 참고해두면 좋습니다

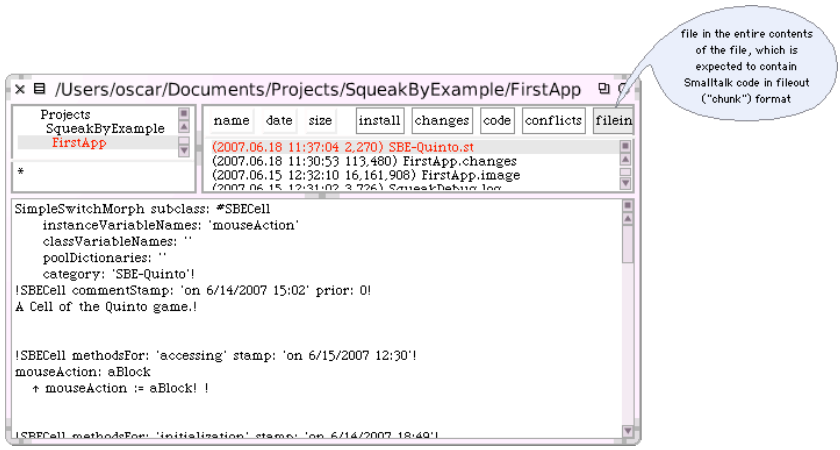


그림 2.14: 스퀵소스 코드 파일에 넣기

몬티첼로 (Monticello) 패키지

비록 파일로 내보내기가 여러분이 작성한 코드의 스냅샷을 만드는 편리한 방법이지만, 이 방법은 확실히 "구식" 입니다. 대부분의 오픈소스 프로젝트를, CVS[6] 또는 Subversion[7]을 사용하여 저장소에서 그들의 코드를 유지하는 작업이 훨씬 더 편리한 것처럼, 스퀵프로그래머도, 몬티첼로*Monticello* 패키지를 사용하여 코드를 관리하는 것이 훨씬 더 편리합니다. 이러한 패키지들은 .mcz 로 끝나는 이름을 가진 파일로 표현하며, 실제로는 패키지의 모든 코드를 zip 으로 압축한 번들입니다.

몬티첼로 패키지 브라우저를 사용하여, FTP와 HTTP 서버를 포함한 다양한 형식의 서버에 있는 저장소에 패키지를 저장할 수 있으며, 또한 로컬 파일 시스템 디렉터리에서 패키지를 바로 작성할 수 있습니다. 패키지의 사본은 항상 로컬 하드 디스크의 package-cache 폴더에 캐쉬합니다. 몬티첼로*Monticello* 는 여러분의 프로그램에 대해 다양한 버전을 저장하고, 버전을 병합하며, 이전 버전으로 되돌리고, 버전간의 차이를 탐색할 수 있게 합니다. 사실 몬테첼로는 분산 리비전 관리 시스템이며, 이는 몬테첼로가 ~~CVS 또는 서브버전~~ git 또

[6] http://www.nongnu.org/cvs
[7] http://subversion.tigris.org

는 Mercurial의 경우처럼, 개발자들의 작업을 다양한 장소에 저장할 수 있게 해준다는 의미입니다.

이메일로 mcz 파일을 보낼 수도 있습니다. 수신자는 *package-cache* 폴더에 저장할 수 있으며, 이 파일을 불러오고 검색하기 위해 몬티첼로*Monticello* 를 사용할 수 있습니다.

 World ▷ open⋯ ▷ Monticello browser 를 선택하여 몬티첼로 브라우저를 여십시오.

브라우저의 오른쪽 창을 보시면 (그림 2.15 를 보십시오) 여러분이 사용하고 있는 이미지에 불러온 코드의 모든 저장소를 포함한 몬티첼로 저장소들의 목록을 보실 수 있습니다.

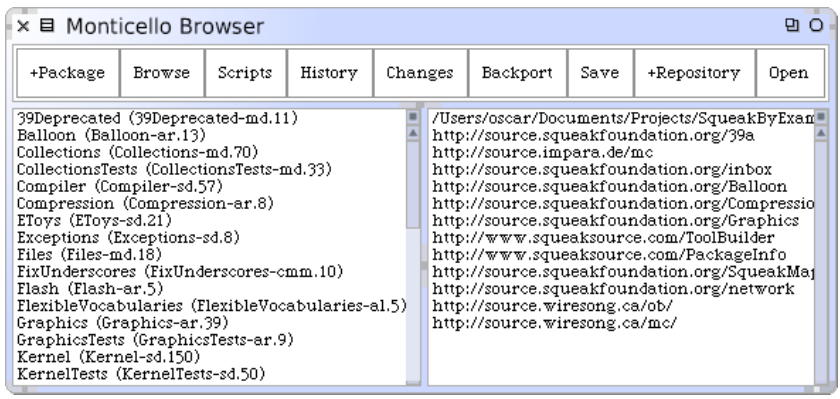


그림 2.15: 몬티첼로 브라우저

몬티첼로 브라우저에서 우측 최상단의 디렉터리는 네트워크를 통해 불러오고 내보낸 패키지의 사본을 캐시 하는 *package-cache* 로컬 디렉터리입니다. 이 로컬 캐쉬는 유용한데, 로컬 기록을 유지해주기 때문입니다. 인터넷 접근할 수 없거나, 원격 저장소에 자주 저장하고 싶지 않을 정도로 연결이 상당히 느린 곳에서도 작업할 수 있게 해줍니다.

몬티첼로에서 코드 저장하고 불러오기

몬티첼로 왼쪽을 보시면 이미지로 불러온 버전 정보를 포함한 패키지 목록이 있습니다. 불러온 이후로 수정된 패키지는 별표(*)로 표시합니다(때때로 이러한 패키지들은 dirty package 로 일컬어지기도 합니다). 패키지를 선택하면 저장소의 목록은 선택한 패키지의 사본을 포함하는 저장소로 제한됩니다.

패키지는 뭘까요? 동일한 접두사를 공유하는 클래스와 메서드 카테고리의 그룹으로 생각할 수 있습니다. Quinto 게임의 모든 코드를 SBE-Quinto 라 부르는 클래스 카테고리에 모두 집어 넣었기 때문에, `SBE-Quinto` package로 그 패키지를 지칭할 수 있습니다.

 `SBE-Quinto` 패키지를 +Package 버튼을 사용하고 `SBE-Quinto`를 입력하여 몬티첼로*Monticello* 브라우저에 `SBE-Quinto` 패키지를 추가하십시오.

SqueakSource: 스퀵용 소스 포지 (Source Forge)

코드를 저장하고 공유하는 최고의 방법은 SquakSource(스퀵소스) 서버에 프로젝트를 위한 계정을 만드는 것이라고 생각합니다. SquakSource는 SourceForge[8]를 닮았습니다. 프로젝트를 관리해주는 웹 프론트엔드 HTTP 몬티첼로 서버입니다. http://www.squeaksource.com[9] 에 공용 스퀵소스 서버가 있으며, 이 책에 관련된 코드 사본은 http://www.squeaksource.com/SqueakByExample.html에 저장되어 있습니다. 웹브라우저로 이 프로젝트를 볼 수 있지만, 자신의 패키지를 관리할 수 있도록 해주는 몬티첼로 브라우저를 사용하여 스퀵내부로부터 작업을 하는 것이 훨씬 더 생산적입니다.

 웹 브라우저에서 http://www.squeaksource.com 을 여십시오. 여러분의 계정을 만들고 *Quinto* 게임에 대한 프로젝트를 만드십시오(예를 들어, *"register"*).

[8] http://www.sourceforge.net

[9] 현시점에서는 새 프로젝트 생성은 http://ss3.gemstone.com/을 이용하라고 안내가 되고있습니다(20130220)

SqueakSource는 몬티첼로 브라우저를 사용하여 저장소를 추가했을 때, 활용할 정보를 보여줄 것입니다.

SqueakSource에 프로젝트를 만들었으면, 스퀵시스템에 이 저장소를 사용하도록 알려야합니다.

 `SBE-Quinto` 패키지를 선택했으면, 몬티첼로 브라우저에서 +Repository 버튼을 클릭하십시오.

사용할 수 있는 여러가지 다른 형식의 저장소가 목록에 보일 것입니다. SqueakSource 저장소를 추가하려면 `HTTP`를 선택하십시오. 서버에 대한 필요한 정보가 대화 상자에 뜰 것입니다. SqueakSource 프로젝트를 인식시키려면 나타난 템플릿을 복사하고 몬티첼로에 넣은 다음 서명과 암호를 기재하십시오:

```
MCHttpRepository
    location: 'http://www.squeaksource.com/!\emph{YourProject}!'
    user: '!\emph{yourInitials}!'
    password: '!\emph{yourPassword}!'
```

비어있는 서명과 비어있는 암호를 사용해도 프로젝트를 불러올 수 있지만 업데이트할 수는 없습니다:

```
MCHttpRepository
    location: 'http://www.squeaksource.com/!\emph{YourProject}!'
    user: ''
    password: ''
```

이 템플릿을 accept 하면 몬티첼로 브라우저 오른쪽에 여러분의 새 저장소가 뜹니다.

 *SqueakSource*에서 *Quinto* 게임의 첫 번째 버전을 저장하기 위해 Save 버튼을 누르십시오.

이미지에 패키지를 불러오려면, 먼저 특정 버전을 선택해야 합니다. 이 동작은 Open 버튼 또는 노랑-버튼 메뉴를 사용하여 열 수 있는 저장소 브라우저에서 수행할 수 있습니다. 버전을 선택하면 이미지에 해당 버전을 불러올 수 있습니다.

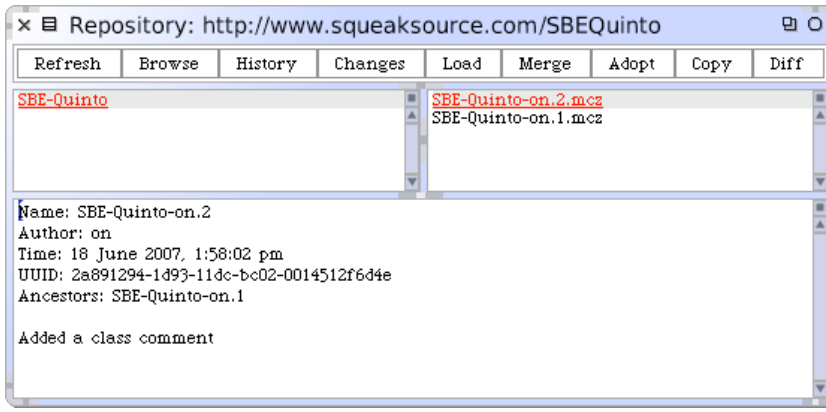


그림 2.16: 몬티첼로 저장소 탐색

방금 저장한 `SBE-Quinto` 저장소를 여십시오.

몬티첼로는 6장 에서 더 깊게 알아볼 수 있는 많은 기능이 있습니다. http://www.wiresong.ca/Monticello/에서 몬티첼로 온라인 문서를 찾아보실 수 있습니다.

2.10 2장 요약

이 장에서 카테고리, 클래스, 메서드를 만드는 방법을 보았습니다. 시스템 브라우저, Inspector, 디버거 그리고 몬티첼로 브라우저의 사용법을 보았습니다.

- 카테고리는 관련 클래스의 묶음입니다.
- 새 클래스는 super클래스에 메시지를 보내서 만듭니다.
- 프로토콜은 관련 메서드의 묶음입니다.
- 새 메서드는 브라우저에서 정의*definition*를 편집하여 만들거나 수정하며, 바뀐 내용은 *accept* 합니다.

- Inspector는 임시 객체를 점검하고 임시 객체와 상호작용하는 간단한 범용 GUI를 제공합니다.
- 시스템 브라우저는 선언하지 않은 메서드와 변수의 사용을 감지하며, 이것들을 사용할 수 있도록 수정 기능등을 제공합니다.
- initialize 메서드는 스퀵에서 객체를 만든뒤, 자동으로 실행됩니다. 이곳에 임의의 초기화 코드를 넣을 수 있습니다.
- 디버거는 실행 중인 프로그램의 상태를 점검하고 수정하기 위한 고급 레벨의 GUI를 제공합니다.
- 카테고리에 정리하고 모아둔 코드를 공유할 수 있습니다.
- 코드를 공유하는 더욱 바람직한 방법은, 외부 저장소를 관리하는 몬티첼로를 사용하는 것입니다. 예를 들어, SqueakSource 프로젝트를 정의하듯이 말이죠.

제 3 장

간단하게 알아보는 문법

스퀵은 최근의 스몰토크규약인 Smalltalk-80 과 매우 유사한 문법을 채택했습니다. 사용되는 문법은 프로그램 문장을 쉬운 영어(pidgin English)처럼 쉽게 읽을 수 있도록 설계되어 있습니다.

```
(Smalltalk includes: Class) ifTrue: [ Transcript show: Class
    superclass ]
```

스퀵은 문법을 최소화 했습니다. 본질적으로 *메시지 보내기*를 위한 문법만 존재합니다. (예, 문법) 문법은 매우 적은 수의 기본적 요소로[1] 만들어집니다. 키워드는 여섯 개 밖에 없으며, 제어 구조 또는 새로운 클래스를 선언하기 위한 문법은 없습니다. 대신, 대부분의 일은 객체에 메시지를 보내 처리합니다. 예를 들어, `if-then-else` 제어 구조 대신에, 스몰토크에서는 `ifTrue:` 와 같은 메시지를 `Boolean` 객체로 보냅니다. 새로운 서브클래스는 상위 클래스로 메시지를 보내어 만들어집니다.

[1] Primitive-컴퓨터 그래픽스에서 그래픽스 프로그램에서 개별적인 실체로 그리고 저장, 만드는 데 필요한 최소조작형을 의미합니다

3.1 문법 요소

문법은 다음의 구성 요소로 이루어져 있습니다:

(i) 여섯개의 준비된 키워드 또는 의사 변수들: self, super, nil, true, false, 그리고 thisContext,
(ii) 숫자, 문자, 문자열, 심볼 그리고 배열을 포함하는 리터럴 객체 [2]에 대한 상수 표현식
(iii) 변수 선언,
(iv) 할당문,
(v) 블록 클로저,
(vi) 메시지 입니다.

표 3.1 에서 다양한 문법 요소의 예를 볼 수 있습니다.

지역 변수 `startPoint`는 변수 이름 또는 식별자 이름입니다. 관례상, 식별자는 "카멜케이스(camelCase)" 로 단어를 구성합니다 (예를 들어, 각각의 단어는 처음 시작하는 단어를 제외하고 첫 글자를 대문자로 시작합니다). 인스턴스 변수, 메서드, 블록 인자, 임시 변수의 첫 번째 글자는 반드시 소문자여야 합니다. 이렇게 선언된 변수의 범위는 클래스 내에서 `private`으로 한정된다는것을 알아야 합니다.

공유 변수 대문자로 시작되는 식별자는 전역 변수, 클래스 변수, pool dictionary 또는 클래스 이름입니다. Transcript는 전역 변수이며, `TranscriptStream`클래스의 인스턴스 입니다.

수신자 self는 실행 중인 현재 메서드 안에 있는 객체를 참조하는 키워드입니다. 이 객체는 보통 실행해야 할 메서드에서 발생한 메시지를 받으므로 이 키워드를 "수신자" 라고 합니다. 우리가 self 에 할당을 할 수 없으므로 의사 변수라고 합니다.

[2] literal - 프로그램 언어에서 문자열 그 자체가 값을 나타내는 것. 예를 들어, X'90' 에서 문자열 90은 90이라는 값을 나타내는 리터럴이다

Syntax	What it represents
startPoint	변수 이름
Transcript	전역 변수 이름
self	의사 변수
1	십진 정수
2r101	이진 정수
1.5	부동 소숫점 수
2.4e7	지수 표기
$a	문자 'a'
'Hello'	문자열 "Hello"
#Hello	심볼 #Hello
#(1 2 3)	리터럴 배열
{1. 2. 1+2}	동적 배열
"a comment"	주석
| x y |	변수 x 와 y 선언
x := 1	1을 x에 할당
[x + y]	x+y을 계산하는 블록
⟨primitive: 1⟩	가상 머신 프리미티브 또는 어노테이션
3 factorial	단항 메시지
3+4	이항 메시지
2 raisedTo: 6 modulo: 10	키워드 메시지
↑true	true값 반환
Transcript show: 'hello'. Transcript cr	표현식 구분자 (.)
Transcript show: 'hello'; cr	메시지 종속 (;)

표 3.1: 간단하게 알아보는 스퀵문법

정수 스퀵은 42와 같은 보통 10진수 정수 뿐만 아니라, 기수 표기법도 제공합니다. 2r101은 2진수의 101이며 [예를 들어, 바이너리], 10진수의 5와 동일합니다.

부동 소수점 수 10진 기반 지수와 함께 지정할 수 있습니다: 2.4e7 is 2.4×10^7.

문자 달러 기호는 리터럴 문자임을 의미합니다: `$a`는 `'a'` 에 대한 리터럴입니다. 비 출력 문자의 인스턴스는 적절한 이름으로 만든 Character space와 Character tab과 같은 적절하게 이름 붙인 메시지를 보내어 Character 클래스에서 비출력 문자에 대한 인스턴스를 가져올 수 있습

니다.

문자열 작은 따옴표 (Single quotes 혹은 게발톱 따옴표) 는 리터럴 문자열을 정의할 때 사용합니다. 문자열 안에서 따옴표를 사용하려면, `'G''day'` 의 경우와 같이 따옴표를 두 개를 넣으면 됩니다.

심볼 문자의 연속 요소를 포함한다는 점에서 문자열과 비슷합니다. 그러나 문자열과 달리, 리터럴 심볼은 전체적으로 유일해야 합니다. Symbol 객체 `Hello`는 하나밖에 없지만, 값이 `'Hello'`인 여러가지 다른 문자열 객체가 있을 수도 있습니다.

컴파일 시점 배열 컴파일 시간 배열은 `#( )`와 공백으로 구분한 리터럴로 둘러쌓아 정의합니다. 괄호 안의 모든 요소는 컴파일 시간 상수여야 합니다. 예를 들어, `#( 27#(true false)abc)`는 세가지 요소의 리터럴 배열입니다: 정수 27, 두 개의 `Boolean` 값을 지닌 컴파일 시간 배열, 그리고 심볼 `abc`가 들어있습니다.

런타임 시점 배열 중괄호 `{ }`는 실행 시간 동안 동적 배열을 정의합니다. 내부 요소는 구두점으로 구분한 프로그램식이 됩니다. 따라서 `{1.2. 1+2}`는 1, 2, 그리고 1+2를 처리한 결과 요소가 들어있는 배열을 정의합니다. (중괄호 기호는 스몰토크스퀵환경의 독자적인 것입니다. 다른 스몰토크에서는 동적 배열을 확실히 만들어야 합니다.)

주석 주석은 큰 따옴표로 둘러싸여 있습니다. 문자열이 아니라 "hello"는 주석이며, 스퀵컴파일러가 무시합니다. 주석은 여러 줄에 걸쳐 있을 수 있습니다.

지역 변수 정의 수직 바는 `| |` 메서드 (그리고 블록 내에서도) 의 하나 이상의 변수 선언을 둘러쌉니다.

할당 `:=` 는 객체를 변수에 할당합니다. 때로는, _를 대신 사용하는 모습을 볼 수 있습니다. 안타깝게도, 이 기호가 ASCII 문자가 아니기 때문에, 특별한 글꼴을 쓰기 전에는 밑줄 문자 처럼 나타납니다. 따라서 `x:=1`은 `x1`, `x_1` 또는 `x` 밑줄 1과 같습니다. 스퀵3.9 부터 다른 표현 방법을 낡음처리 (deprecated) 해놓았으므로 `:=` 를 사용해야 합니다.

블록 각괄호 [ ]는 블록을 정의하며, 함수를 나타내는 첫 번째 Object 클래스인 블록 구분자*block closure* 또는 어휘 구분자*lexical closure* 로 알려져 있습니다. 우리가 살펴볼 내용처럼, 블록은 인수를 취하거나 로컬변수를 가질 수 있습니다.

프리미티브 `<primitive:\ldots>`는 가상 머신 프리미티브의 Call을 나타냅니다. (`<primitive: 1>`는 `SmallInteger>>+`)를 위한 가상 머신 프리미티브를 가리킵니다. 이 프리미티브 다음의 코드는 프리미티브 처리에 실패했을 경우에만 실행합니다. 스퀵3.9 부터는, 메서드 어노테이션에 동일한 문법을 사용합니다.

단항 메시지 (3과 같은) 수신자로 보내는 (factorial과 같은) 한 개의 단어로 구성됩니다.

바이너리 메시지 (+와 같이) 수신자로 보내며 단일 인자를 취하는 연산자입니다. 3+4에서 수신자는 3이며 인자는 4입니다.

키워드 메시지 (`raisedTo:modulo:`와 같은) 여러가지 키워드로 구성되어 있으며, 각각 콜론으로 끝나고, 단일 인자를 취합니다. raisedTo: 6 modulo: 10 이라는 프로그램식에서, 메시지 셀렉터 `raisedTo:modulo:`는 6과 10 두 개의 인자를 취하는데, 각각의 인자는 콜론 뒤에 붙어있습니다. 이 메시지를 수신자 2로 보냅니다.

메서드 반환 ^는 메서드에서 값을 반환할 때 사용합니다. (^ 문자를 쓰려면^을 입력해야 합니다.)

선언문 순차 배치 마침표는 (.) 선언문 구분자입니다. 두개의 프로그램식 사이에 마침표를 넣으면, 독립된 선언문으로 바뀝니다.

종속 세미콜론은 종속된 여러 메시지를 단일 수신자에 보내는데 사용할 수 있습니다. `Transcript show: 'hello'; cr`에서 `show: 'hello'` 키워드 메시지를 수신자 Transcript에 먼저 보내고, 단항 메시지 cr를 동일한 수신자에 보냅니다.

`Number`, `Character`, `String`, `Boolean` 클래스는 8장 에서 상세하게 다루겠습니다.

3.2 가상 변수

스몰토크에서는, 여섯개의 예약어가 있는데 이를 가상 변수*Psudo-valiables* 라고도 합니다:

self, super, nil, true, false, thisContext 가 바로 그것입니다. 가상 변수라고 부른 이유는 미리 정의하거나 할당할 수 없기 때문입니다.[3] 코드가 수행되는 중에 `self`, `super`, `thisContext`의 값은 코드를 실행하면서 동적으로 바뀌는 반면에 `true`, `false`, `nil`은 애초부터 정의되어있는 상수입니다.

`true`와 `false`는 Boolean 클래스인 True 와 False 의 고유한 인스턴스 입니다. 더 자세한 내용은 8장 을 참조하십시오.

`self`는 항상 현재 메서드를 실행 중인 수신자를 참조합니다.

`super`는 항상 현재 메서드의 수신자를 참조하지만, 메시지를 super 로 보내면, 메서드 검색 위치가 바뀌어서, `super`를 사용하는 메서드를 포함하는 클래스의 super클래스에서 시작합니다. 더 자세한 내용은 5장 을 참조하십시오.

`nil` 은 정의하지 않은 오브젝트를 의미합니다. 이것은 클래스 UndefinedObject의 고유한 인스턴스입니다. 인스턴스 변수, 클래스 변수, 지역 변수는 nil로 초기화 합니다.

`thisContext`는 런타임 시점에서 스택의 최상위 프레임을 나타내는 가상 변수입니다. 다르게 말하자면, 현재 실행 중인 MethodContext 또는 BlockContext를 나타냅니다. thisContext는 보통 대부분의 프로그래머들이 관심을 가지지 않지만, 디버거와 같은 개발 도구를 실행하기 위해 필수적이며,

[3] 미리 정의하거나 할당될 수 없다는건 클래스의 인스턴스가 생성되는 시점에서 자동으로 존재될 수 있기 때문입니다. 새로 값을 할당하거나 하는게 아니라 런타임시에 동적으로 자동으로 할당되기 때문인거죠. 예를 들자면 self의 경우는 인스턴스 내부에서 자신을 가리키는 변수기때문에 별도로 정의하거나 할 수 있는게 아니라는 의미입니다.

예외 처리및 계속 실행을 구현하기 위해 사용하기도 합니다.[4]

3.3 메시지 보내기

스퀵에는 3가지 종류의 메시지가 있습니다.

1. *단항* 메세지는 인자를 취하지 않습니다. `1 factorial`은 메시지 `factorial`을 `1`이라는 객체에 보냅니다.
2. *이항* 메시지는 정확히 하나의 인자를 취합니다. `1+2`는 인자 `2`와 함께 메시지 `+`를 `1`이라는 객체에 보냅니다.
3. *키워드* 메시지는 임시로 여러 개의 인자를 취합니다. `2 raisedTo:6 modulo: 10`은 메시지 셀렉터 `raisedTo:modulo:`와 인자 `6`과 `10`으로 구성된 메시지를 `2`라는 객체에 보냅니다.

단항 메시지 셀렉터는 알파벳-숫자 문자로 구성되어 있으며, 소문자로 시작합니다.

이항 메시지 셀렉터는 다음 집합의 하나 이상의 문자로 이루어져 있습니다:

```
+ -- / \ * ~ < > = @ % | & ! ? ,
```

키워드 메시지 셀렉터는 소문자로 시작하며 콜론으로 끝나는 알파벳 숫자 키워드의 모음으로 구성되어 있습니다.

단항 메시지는 가장 높은 우선순위를 가지고 있으며, 그 다음 이항 메시지고, 마지막이 키워드 메시지 입니다. 따라서:

```
2 raisedTo: 1 + 3 factorial  ⟶  128
```

[4] http://blog.naver.com/PostView.nhn?blogId=tkandrea92&logNo=80013761680&parentCategoryNo=4&viewDate=&currentPage=1&listtype=0 이 부분에 있는 설명이 조금 더 정확하다고 생각됩니다. 현재 실행되고있는 프레임의 주소값. 그러니 디버깅할때 의미가 있겠죠.

(먼저 우리는 `factorial`을 `3`에 보내고, `+6`을 `1`로 보낸다음, `raisedTo:7`을 `2`에 보냅니다.[5]) 앞서 배운, 기호 expression-->result를 수식 계산 결과를 보여주기 위해 사용한다는 것을 상기해 주시기 바랍니다.

우선 순위에 따라, 처리는 엄격하게 왼쪽에서 오른쪽으로 진행합니다. 따라서 아래 계산식의 결과는

```
1 + 2 * 3  ⟶  9
```

`7`이 아닙니다. 계산의 순서를 바꾸려면 괄호를 사용해야 합니다:[6]

```
1 + (2 * 3)  ⟶  7
```

메시지 보내기는 마침표와 세미콜론으로 작성합니다. 마침표로 구분한 구문의 순차 배치는 선언문 과 같이 개별로 나뉘어진 구문을 순차적으로 처리하도록 합니다.

```
Transcript cr.
Transcript show: 'hello world'.
Transcript cr
```

위의 코드는 `cr`을 Transcript 객체로 보낸 후, `show: 'hello world'`에 보내고, 마지막으로 다른 `cr`을 보냅니다.

여러가지 메시지를 동일한 수신자로 보내고 있을 때, 일련의 과정들은 종속 구조와 같이 더욱 간결하게 표현될 수 있습니다. 수신자는 한 번 만 구체적으로 지정하며, 메시지들의 각 시퀀스는 세미콜론으로 구분합니다:

```
Transcript cr;
    show: 'hello world';
    cr
```

이것은 앞의 예제와 정확히 같은 결과를 냅니다.

---

[5] 해당되는 문장을 조금 더 자세히 분석한 내용은 이곳을 참고하시면 됩니다.

[6] raisedTo: 같은 셀렉터의 경우는 연산자보다 우선순위가 뒤로 밀립니다. 일반적인 사칙연산셀렉터가 우선순위가 높은데 다른예의 factorial의 경우는 3이라는 객체에 대한 단항 셀렉터이기때문에 객체 단위가 하나로 잡혀서 우선적으로 계산되는 결과가 나오는겁니다. 3 factorial은 3을 그대로 전달하는것같은 하나의 단위가 된다는 의미죠.

3.4 메서드 문법

스퀵에서 구문은 어디서든 처리될 수 있어야 하지만 (예를 들어, Workspace, 디버거 또는 브라우저), 메서드는 보통 브라우저 창 또는 디버거등에서 정의됩니다. (메서드는 외부 장치에 파일로 저장할 수 있지만, 보통 스퀵에서 프로그래밍하는 방법은 아닙니다.)

프로그램은 주어진 클래스의 상태정보등에 따라 한번에 하나의 메소드씩 개발합니다.(클래스는 기존의 클래스로 메시지를 보내어 서브클래스를 만들도록 요청하여 정의합니다. 따라서, 클래스를 정의하는데 특별한 문법이 필요한 것은 아닙니다.)

`String` 클래스의 `LineCount` 메서드를 살펴보겠습니다. (보통의 관례에서는 `ClassName>>methodName`와 같이 메서드를 참조하므로, 이 메서드를 `lineCount` 라고 부르겠습니다.)

Method 3.1: 줄 수 세기

```
String>>lineCount
   "Answer the number of lines represented by the receiver,
   where every cr adds one line."
   | cr count |
   cr := Character cr.
   count := 1 min: self size.
   self do:
      [:c | c == cr ifTrue: [count := count + 1]].
   ↑ count
```

메서드의 각 문법적 구성 요소는 다음과 같습니다:

1. 이름(예를 들어, `lineCount`)과 몇 가지 인자(이 예에서는 아무것도 없습니다)를 포함하는 메서드 패턴;
2. 주석(어디서든 있을 수 있지만, 관례상 메서드가 무슨 일을 하는지 맨 위에 적습니다);
3. 지역 변수 정의(예를 들어, `cr`과 `count`)의 선언, 그리고
4. 점으로 구분한 수많은 구문이 있습니다.

^(^로 입력) 보다 우선되는 구문의 처리는 메서드를 빠져나갈 시점에 표현식의 값을 반환합니다. 어떤 구문의 결과 값을 분명하게 반환하지 않고 끝나는 메서드는 암묵적으로 self 를 반환합니다.

인수와 지역 변수는 항상 소문자로 시작합니다. 대문자로 시작하는 이름은 전역 변수로 간주합니다. 예를 들어, `Character`와 같은 종류의 클래스 이름은, 그 클래스를 나타내는 오브젝트를 참조하는 간단한 전역 변수임을 나타냅니다.

3.5 블록 표현식

블록은 프로그램식의 처리를 결정하는 메커니즘을 제공합니다. 블록은 본질적으로 익명 함수입니다. 블록은 value 메시지를 보내 처리합니다. 블록은 확실한 반환문(^로 입력) 없이 어떠한 값도 반환하지 않으면, 블록 자체 내의 마지막 프로그램식의 값으로 반환됩니다.

```
[ 1 + 2 ] value  ⟶  3
```

블록은 매개 변수를 가지며, 각각의 매개 변수는 콜론의 뒤를 따르며 선언됩니다. 수직 바(pipe문자)는 블록의 내용과 매개 변수 선언을 분리합니다. 하나의 매개 변수를 가진 블록을 처리하려면, 하나의 인자를 담은 value: 메시지를 보내야 합니다. 두 개의 매개 변수를 가진 블록은 `value:value:` 를 보내야 하며, 이런 식으로 4개까지 인자를 처리 할 수 있습니다.[7]

```
[ :x | 1 + x ] value: 2  ⟶  3
[ :x :y | x + y ] value: 1 value: 2  ⟶  3
```

블록에 4개를 넘는 매개 변수가 있다면, 반드시 `valueWithArguments:` 를 사용해야 하고 배열로 전달해야 합니다. (많은 수의 파라미터를 가진 블록은 종종 설계 문제의 원인이 됩니다)

[7] [:x :y :aa :cc | x + y + (aa * cc)] value: 1 value: 2 value: 3 value:4 대략 이런식으로 4개의 인수를 쓴다는 의미

블록은 메서드의 지역 변수 선언과 같이, 수직 바로 둘러쌓은 지역 변수를 선언하기도 합니다. 지역 변수는 인자 다음에 선언합니다.

```
[ :x :y | | z | z := x+ y. z ] value: 1 value: 2  ⟶  3
```

블록은 실제로 어휘 구분자*lexical closure* 이며, 주변 환경의 변수들을 참조할 수 있습니다. 다음 블록은 그 블록과 가까운 환경의 변수 x를 참조합니다:

```
| x |
x := 1.
[ :y | x + y ] value: 2  ⟶  3
```

블록은 클래스 BlockContext 의 인스턴스 입니다. 이것이 의미하는것은, 블록은 오브젝트이므로, 변수에 할당될 수 있고, 모든 다른 오브젝트와 똑같이 인수로서 전달될 수 있다는 것입니다.[8]

충고: 스퀵의 현재 버전 (3.9)은 트루 블록 클로저(true block-closure)를 실제 지원하지 않으며, 그 이유는 블록 인수가 인클로징 메서드의 임시 변수로서 실제적으로 시뮬레이션 되기 때문입니다. 완전한 블록 구분자full block closure 를 지원하는 새로운 컴파일러가 있지만, 그 컴파일러는 여전히 작업중이며, 기본적으로 사용되지는 않습니다.

대부분의 애매한 상황에서, 이 문제는 이름 충돌*naming conflict* 을 유발할 수 있습니다. 이 상황은 스퀵이 스몰토크의 이전 구현을 기반으로 하기 때문에 발생합니다. 이런 여러 가지 문제에 직면하혔다면, 메서드 `fixTemps` 의 sender 를 보시거나 클로저 컴파일러 *the closure compiler* 를 불러오시기 바랍니다.

3.6 간단하게 살펴보는 조건과 루프 (Conditionals and loops in a nutshell)

스몰토크는 제어문을 만들기 위한 특별한 문법을 제공하지 않습니다. 그대신, 제어문은 인수로서의 블록과 함께, `Boolean`, 숫자 그리고 컬렉션에 메시지를

[8] 블록 연산자체도 인수로 전달될 수 있다는 의미입니다. http://workspace.onionmixer.net/mediawiki/index.php?title=Smalltalk_Tips 이 페이지의 "block을 이용한 인자의 처리에 대한 Transcript와 string출력" 부분을 참고해주세요

보내는 표현을 이용해서 만들수 있습니다.

조건들은 메시지 `ifTrue:`, `ifFalse:` 또는 `ifTrue:ifFalse:` 들 중 하나를 `Boolean` 표현식의 결과쪽으로 보내서 표현합니다. `Boolean`에 대해 좀 더 많은 내용을 원하시면 8장 을 보십시오.

```
(17 * 13 > 220)
   ifTrue: [ 'bigger' ]
   ifFalse: [ 'smaller' ]  ⟶  'bigger'
```

반복문은 일반적으로 블록, 정수 또는 컬렉션에 메세지를 보내는것으로 표현됩니다. 루프의 종료 조건을 반복적으로 판단하기 때문에, 루프는 `Boolean` 값보다는 블락이 되어야 합니다. `Boolean`은 종료조건이 한번에 끝나버릴 수 있기 때문이죠. 여기에 절차적 반복문에 대한 예가 있습니다:

```
n := 1.
[ n < 1000 ] whileTrue: [ n := n*2 ].
n  ⟶  1024
```

`whileFalse:` 나가기 조건을 뒤바꿉니다.

```
n := 1.
[ n > 1000 ] whileFalse: [ n := n*2 ].
n  ⟶  1024
```

`timesRepeat:` 고정회수 반복을 실행하기 위한 간단한 방법도 제공됩니다:

```
n := 1.
10 timesRepeat: [ n := n*2 ].
n  ⟶  1024
```

메시지 `to:do`를 반복 카운터의 초기 값 역할을 하는 숫자에 보낼 수 있습니다. 아래의 예제에서 볼 수 있듯이 `to:do` 에서 두 개의 인수는 상한선이 되는 값과 반복 카운터의 현재 값을 취하는 블록을 자신의 인수로 받습니다:

```
result := String new.
1 to: 10 do: [:n | result := result, n printString, ' '].
result  ⟶  '1 2 3 4 5 6 7 8 9 10 '
```

오름차순 반복 처리자 (High-order Iterators)

컬렉션들[9]은 수 많은 다양한 클래스로 이루어져 있으며, 그들 중 대부분의 클래스는 동일한 프로토콜을 지원합니다. 이러한 컬렉션들은 반복 처리를 지원하기 위한 몇가지 중요한 셀렉터를 공통으로 가지고 있는데 가장 중요한 메시지로 `do:`, `collect:`, `select:`, `reject:`, `detect:`, `inject:`, `into:` 등이 포함됩니다. 이러한 메시지들은, 고수준의 압축된 코드를 정의할 수 있도록 고수준의 반복처리자*high-level iterators* 를 사용할 수 있게 합니다.[10]

인터벌 *Interval* 은 시작부터 끝까지 숫자의 시퀀스를 반복처리 해주는 컬렉션입니다. `1 to:10`은 1에서 10까지의 인터벌을 나타내며, 메시지 do:를 인터벌에 보낼 수 있습니다. 인수는 컬렉션의 각 구성요소를 위해 계산되는 블록입니다.

```
result := String new.
(1 to: 10) do: [:n | result := result, n printString, ' '].
result  ⟶  '1 2 3 4 5 6 7 8 9 10 '
```

`collect:` 각 구성요소를 변화시켜, 동일한 크기의 새로운 컬렉션을 만듭니다.

```
(1 to: 10) collect: [ :each | each * each ]  ⟶  #(1 4 9 16 25
    36 49 64 81 100)
```

새로운 컬렉션을 만들고, 각각의 컬렉션에 `Boolean` 블록 조건을 만족하는 구성요소의 서브세트*subset* 를 셀렉터에 메세지로 보냅니다. detect:는 조건을 만족시키는 첫 번째 구성요소를 반환합니다. String 또한 컬렉션이므로 모든 Character들은 반복 처리 할 수 있다는 것을 잊지 말아주십시오.[11][12]

---

[9] squeak의 class browser등에서 찾아보면 collections라고 되어있는 많은 패키지들을 볼 수 있습니다. iterator를 이용한 데이터의 탐색을 지원하는 패키지들.

[10] Collection에 대한 조금 더 자세한 내용은 http://ta.onionmixer.net/wordpress/?p=163 내용을 참고

[11] 이 경우 'hello there' 라고 선언된 문자열 자체가 Collection 입니다. 왜냐하면 String 클래스 자체가 Collection을 상속받아 만들어졌기 때문입니다. char라는것에는 의미는 딱히 없습니다. 그냥 Iterator의 변수명이니까요.

[12] isVowel은 Kernel-BasicObjects 패키지의 Chracter라는 class의 셀렉터이며 이 예제에서는 String을 배열로 char라는 Iterator를 이용해서 배열의 앞에서 맨 뒤까지 순차적으로 처리해서 현재 가리키고있는 배열의 element가 모음인경우 해당되는 문자를 반환하는데 사용된다

```
'hello there' select: [ :char | char isVowel ]  ⟶  'eoee'
'hello there' reject: [ :char | char isVowel ]  ⟶  'hll thr'
'hello there' detect: [ :char | char isVowel ]  ⟶  $e
```

마지막으로, 컬렉션이 `inject:into:` 메소드에서 *functionalstyle fold operator*[13]를 지원한다는 것에 주의해야 합니다. 이것은 씨앗값(seed value)으로 시작하고, 컬렉션의 각 구성요소를 넣어주는 표현식을 사용하여 누적된 결과를 발생시킵니다. 합계와 곱셈의 결과는 이런 경우에 대한 전형적인 예입니다.

```
(1 to: 10) inject: 0 into: [ :sum :each | sum + each ]  ⟶  55
```

이 코드는 `0+1+2+3+4+5+6+7+8+9+10`와 같습니다..

컬렉션에 대한 좀 더 많은 내용은 9장 에서 보실 수 있습니다.

3.7 프리미티브와 프라그마

스몰토크에서는 모든 것이 객체이며, 모든 것은 메시지를 보내어 동작합니다. 그럼에도 불구하고, 우리는 특정 지점에서, 한계를 보게 됩니다. 특정 오브젝트는 오직 가상 머신 에서 제공되는 프리미티브 를 불러온 뒤 사용할 수 있습니다.

예를 들어, 다음 오브젝트들은 모두 프리미티브로서 실행됩니다: 메모리 할당(`new`, `new:`), 비트조작(`bitAnd:`, `bitOr:`, `bitShift:`) 포인터와 정수 연산 (`+`, `-`, `<`, `>`, `*`, `/`, `=`, `==`) 그리고 배열 접근. (`at:`, `at:put:`).

프리미티브는 프로그램식인 `<primitive: aNumber>`[14] 형식으로 불러옵니다. 이러한 프리미티브를 불러오는 메서드는 오직 그 프리미티브가 오류가 나는 경우, 오류의 다음에 처리할 스몰토크코드 또한 포함할 수 있습니다.

[13]함수처럼 연산이 가능한 블록으로 쌓인 연산자

[14]VM image에 있는것을 사용하는게 아니라 VM Engine에서 뭔가를 불러올때 사용합니다. 조금 다른 사용예를 보고싶으면 여기를 참고해주세요

여기에 우리는 SmallInteger>>+[15]를 위한 코드를 볼 수 있습니다. 만약 프리미티브가 오류가 나면 표현식 super+aNumber를 계산하고 반환합니다.

Method 3.2: 프리미티브 메서드(SmallInteger 클래스의 + 셀렉터 선언부)

```
+ aNumber
    <primitive: 1>
  ↑ super + aNumber
```

스퀵3.9 이후로, 꺽쇠 괄호 문법*the angle bracket syntax* 은 프라그마*pragma*[16]라고 불리는 메서드 어노테이션*method annotation* 에 사용합니다.

3.8 3장 요약

- 스퀵은 가상 변수라고 불리기도 하는 여섯개의 예비 식별자를 갖고 있습니다:
  `true`, `false`, `nil`, `self`, `super`, `thisContext`.
- 다섯 종류의 리터럴 오브젝트가 있습니다: 숫자(`5`, `2.5`, `1.9e15`, `2r111`), 문자(`$a`), 문자열 (`'hello'`), 심볼(`#hello`), 그리고 배열(`#('hello'#hello)`)
- 문자열은 작은 따옴표로 범위를 정하며, 주석은 큰 따옴표로 정합니다. 문자열 내부에서 사용하려면 큰 따옴표를 사용합니다.
- 문자열과 달리, 심볼은 광범위한 고유성을 보장받습니다.
- 리터럴 배열을 정의하기 위해 `#( ... )`를 사용합니다. 동적 배열을 정의하기 위해 `{ ... }`를 사용합니다. `#( 1 + 2 ) size --> 3` 이지만 `{ 1 + 2 } size --> 1`임에 유의합니다.

[15] SmallInteger 클래스의 + 라는 셀렉터를 의미. Smalltalk에서는 연산자도 셀렉터입니다. 해당되는 클래스를 Class Browser에서 찾아보면 좀 더 명확하게 알 수 있습니다

[16] C언어등에서는 소스코드내에서 compiler에 지시를 내리는 지시자등으로 사용됩니다. 자세한 내용은 여기를 참고

- 세 종류의 메시지가 있습니다:
  단항 메시지(예를 들어, `1 asString`, `Array new`), 이항 메시지(예를 들어, `3+4`, `'hi', ' there'`), 키워드 메시지(예를 들어, `'hi' at: 2 put: $o`)

- 캐스케이드 메시지 발송은 세미콜론으로 분리되어 동일한 대상으로 보냅니다:
  ```
  OrderedCollection new add: #calvin; add: #hobbes; size --> 2
  ```

- 지역 변수는 수직 바와 함께 선언합니다. 할당을 위해 := 를 사용합니다. _*To Left Arrow* 또는 _는 작동하지만, 스퀵3.9 이후로, 비추천 기호입니다.
  ```
  |x| x:=1
  ```

- 수식식은 메시지 sends, cascades, assingnments로 구성되며, 괄호로 그룹을 만들 수 있습니다. 선언문은 구두점으로 구분한 수식입니다.

- 블록 구분자*Block closures* 는 꺽쇠 괄호로 둘러싼 수식입니다. 블록은 인수를 취할 수 있고 임시 변수를 포함할 수 있습니다.

- 블록에 있는 수식은 여러분이 블록 값(`value...`) 메시지를 정확한 숫자의 인수와 함께 보낼 때까지 계산하지 않습니다.
  ```
  [:x | x + 2] value: 4 --> 6.
  ```

- 제어 구성을 위한 구문은 없으며, 메시지가 조건적으로 블록을 처리합니다.
  ```
  (Smalltalk includes: Class) ifTrue: [ Transcript show: Class s
  uperclass ]
  ```

제 4 장

메시지 표현식의 이해

비록 스몰토크의 메시지 문법은 매우 단순하지만, 독특하며[1] 사용에 익숙해지기까지 약간의 시간이 걸립니다. 이 장은 여러분이 메시지를 보내기 위해 이 특별한 프로그램식에 익숙해 지도록 돕기 위한 몇 가지 안내 사항을 제공해 드릴 것입니다. 만약 여러분이 프로그램식을 이미 편안하게 느끼신다면, 이 장을 생략하시거나 나중에 다시 보셔도 됩니다.

4.1 메시지 식별하기

스몰토크에서, 3장 에 나열된 `:= ^ . ; # () {} [ : | ]` 문법의 구성요소를 제외한 모든 것이 메시지 전송입니다. C++ 에서 그랬던 것처럼, 사용자 자신의 클래스를 위해 +와 같은 연산자들을 정의할 수 있지만, 모든 연산자들이 동일한 우선권을 가지고 있는 것은 아닙니다. 무엇보다, 사용자는 메서드의 인자를 변경할 수 없습니다. "-"는 항상 이항 메시지 이며, 다른 오버로딩을 통해 단항 "-"을 가질 수 있는 방법은 없습니다.

스몰토크에서 보낸 메시지들의 순서는 메시지의 종류에 의해 결정됩니다. 스-

[1] 원문단어는 Unconventional 입니다. 관례적이지 않은.. 이라는 뜻이 있습니다만.. 여기서는 독특하다는 뜻을 차용하기로 했습니다.

몰토크에는 오직 3가지 종류의 메시지, 즉 단항 메시지, 이항 메시지 그리고 키워드 메시지 가 있습니다. 단항 메시지는 항상 먼저 전송되며, 그 다음 이항 메시지 그리고 마지막으로 키워드 메시지들의 순서로 전송됩니다. 대부분의 언어에서, 괄호 는 계산 순서를 바꾸기 위해 사용합니다. 이러한 규칙들은 스몰토크코드의 독해를 보다 쉽게 만들기도 합니다. 일반적인 경우 사용자가 규칙들에 관해 생각할 필요는 없습니다.

스톨토크에서 대부분의 연산은 메시지 전달로 이루어지기 때문에 메시지를 정확하게 식별하는 작업은 매우 중요합니다. 다음의 용어들을 보면 메시지를 정확하게 식별하는데 도움을 얻을 수 있습니다.

- 메시지는 메시지 선택자 *Selector* 와 추가 메시지 인자로 구성됩니다.
- 메시지는 수신자 *Receiver* 에게 전송됩니다.
- 메시지와 메시지 수신자의 조합을 그림 4.1 에서 보이는 것과 같이 메시지 전송 부(또는 부분)[2]이라고 합니다.

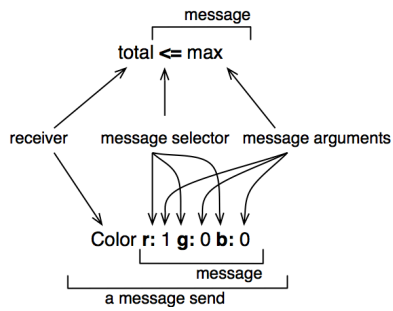


그림 4.1: 수신자와 메서드 셀렉터로 구성된 두 개의 메시지와 인자들의 집합

그림 4.2: Morph color: `Color yellow`는 두 개의 메시지 전송부분으로 구성되어 있습니다: `Color yellow` 와 `Morph color: Color yellow`

> 메시지는 항상 단일문자(문자열)이 될 수 있는 블록, 변수 또는 다른 메시지의 계산결과를 수신자로 보냅니다.

[2] 원문은 message send라고 되어있습니다만한국어로 하면 "메시지 전송" 에서 끝나버리기 때문에 특정 부분을 지칭하는 메시지 전송부 정도로 번역하도록 하겠습니다.

메시지 전송	메시지 형식	결과
Color yellow	단항	색상 만들기
aPen go: 100.	키워드	수신펜(receiving pen)을 100 픽셀 앞으로 이동
100 + 20	이항	숫자 100이 메시지를 수신 + 숫자 20과 함께
Browser open	단항	새로운 브라우저 열기
Pen new go: 100	단항 및 키워드	펜을 만들고 100 픽셀 이동
aPen go: 100 + 20	키워드 및 이항	수신펜(receiving pen)을 120 픽셀 앞으로 이동

표 4.1: 메시지 전송과 그 메시지 형식들의 예

메시지의 수신자를 식별하는걸 돕기 위해, 수신자는 밑줄로 표시되어있습니다. 그리고 타원과 번호로 각 메시지를 영역으로 분리했습니다. 이렇게 하면 진행순서를 파악하는데 도움이 됩니다.

그림 4.2 는 두 개의 메시지 전송, `Color yellow`와 `aMorph color: Color yellow`를 나타냅니다. 그렇기때문에 두 개의 타원이 있습니다. 전송되는 메시지인 `Color yellow` 는 첫 번째로 실행되므로, 이 타원에 번호 `1`을 붙였습니다. 두 개의 수신자가 있는데, 첫 번째는 메시지 `color:`를 수신하는 `aMorph`이며, 다른 하나는 메시지 `yellow` 를 수신하는 `Color`입니다. 두 개의 수신자에 밑줄을 그었습니다.

`100 + 200` 이라는 메시지 전송부의 100 또는 Color Yellow 라는 메시지 전송부의 Color처럼 메시지의 첫째 요소는 수신자가 될 수도 있습니다. 하지만 수신자는 다른 메시지의 결과도 될 수 있습니다. 예를 들어, 표 4.1에서 메시지 Pen new go:100 에서 메시지 go: 100은 Pen new의 결과로 인해 반환된 객체를 수신자로 인식하고 메시지를 보냅니다. 어떤 상황이라도 메시지는 다른 메시지의 결과를 받은 수신자를 메시지로 인식하고 그것을 호출한 객체로 보냅니다.

표 4.1 에서 몇가지 예를 볼 수 있습니다. 모든 메시지 전송부들이 인자를 가져야 하는건 아니라는 것은 아닙니다. open과 같은 단항 메시지는 인자가 없습니다. `go: 100` 그리고 `+ 20` 과 같은 단일 키워드와 이항 메시지는 각각 한 개씩의 인자를 가집니다. 메시지에는 간단한 메시지들과 구성된 메시지등이 있습니다. Color yellow와 `100 + 20`는 간단합니다: 객체로 보낸 `aPen go: 100+20;` 메시지는 두 부분으로 나누어져 있습니다: + 20을 100에게 보낸다

음 이 메시지의 결과를 go:의 인자로 만들어서 aPen에게 보내게 되어있죠. 수신자는 그 자체로 오브젝트가 반환되는 표현(예를 들어, 메시지 또는 문자를 할당하는경우)이 되는 경우도 있습니다. Pen new go: 100 에서 보면 go: 100이라는 메시지는 Pen new라는 메시지가 실행된 결과인 오브젝트로 전송됩니다.

4.2 세 종류의 메시지

스몰토크는 메시지가 전송되는 순서를 결정하기 위해 메시지를 몇가지 간단한 규칙으로 정의합니다. 이러한 규칙들은 세 종류의 메시지구분에 대한 차이를 기반으로 합니다:

- *단항 메시지*는 어떤 정보도 갖고 있지 않은 객체에게 발송한 메시지입니다. 예를 들어, `3 factorial`에서, `factorial`은 단항 메시지 입니다.
- *이항 메시지*는 연산자들(종종 산술적)로 구성되어 있습니다. 그 연산자들은 이항(양쪽에 인자가 필요한)이며, 그 이유는 이항메세지는 항상 두 개의 객체, 그리고 수신자와 인자 객체등에 관여하기 때문입니다. 예를 들어, 10 + 20 에서 +는 인자 20과 함께 수신자 10에 발송된 이항 메시지 입니다.
- *키워드 메시지*는 한 각각 콜론(:)으로 끝나며 한 개의 인자를 받는 한 개 또는 그 이상의 키워드로 구성됩니다. 예를 들어, `anArray at: 1 put: 10`에서, 키워드 at:은 인자 1을 취하며, 키워드 put:은 인자 10을 취합니다.

단항 메시지

단항 메시지는 어떤 인자도 요구하지 않는 메시지 입니다. 단항 메시지는 receiver messageName의 문법구조를 따릅니다. 이런경우에 셀렉터는 그 뒤에 문자(인자)가 이어지지 않게 만들어져 있습니다. (예를 들어, 계승(`factorial`), 열기(`open`), 클래스(`class`)

```
89 sin          ⟶  0.860069405812453
3 sqrt          ⟶  1.732050807568877
Float pi        ⟶  3.141592653589793
'blop' size     ⟶  4
true not        ⟶  false
Object class    ⟶  Object class  ""
```

> 단항 메시지는 어떤 인자도 요구하지 않는 메시지 입니다. 단항 메시지는 receiver **selector**의 문법구조로 되어 있습니다

이항 메시지

이항 메시지는 정확히 하나의 인자를 필요로 하고 메세지의 선택자가 +, -, *, /, &, =, >, |, <, ~, @등과 같은 분류에서 가져온 하나 이상의 문자가 붙어오도록 만들어져 있는 메시지입니다: --은 문장의 해석양식에서 허용되지 않습니다.

```
100@100           ⟶  100@100  "creates a Point object"
3 + 4             ⟶  7
10 -- 1            ⟶  9
4 <= 3            ⟶  false
(4/3) * 3 = 4     ⟶  true  "등식은 이항 메시지이며 분수 표현은
                                                  정확합니다"
(3/4) == (3/4)    ⟶  false "두개의 동일한 분수는 같은 객체가`아닙니다"
```

> 이항 메시지는 정확히 하나의 인자를 요구하고 그의 선택자가 : +, -, *, /, &, =, >, |, <, ~, @등과 같은 분류에서 가져온, 하나 또는 그 이상의 문자의 연속으로 만들어져 있는 메시지입니다. 이항 메시지는 receiver **selector** argument의 문법구조로 되어있습니다.

키워드 메시지

키워드 메세지는 각각 콜론으로 끝나는 셀렉터와 그에 따르는 한개 또는 복수의 인자로 구성된 메세지입니다. 키워드 메시지는 다음의 문법구조로 되어있

습니다: `receiver selectorWordOne: argument-One wordTwo: argumentTwo`

키워드는 따로따로 인자를 취합니다. 그러므로 `r:g:b:`는 3 개의 인자를 가진 메서드이며, `playFileNamed:`와 `at:`은 1개씩의 인자를 가진 메서드고, `at:put:`는 2 개의 인자를 가진 메서드 입니다. `color` 클래스로 인스턴스를 만들면, `Color r:1 g:0 b:0`에서와 같이 색상을 빨강으로 만드는 메서드 `r:g:b:`을 사용할 수 있습니다. `:`(콜론)은 셀렉터의 일부임을 주의합니다.

> 자바 또는 C++에서, 스몰트크 메서드 불러오기. Color r: 1 g: 0 b: 0를 Color.rgb(1,0,0)로 작성할 수 있습니다.

```
1 to: 10               ⟶  (1 to: 10)  "인터벌을 만듭니다"
Color r: 1 g: 0 b: 0   ⟶  Color red   "새 색상을 만듭니다"
12 between: 8 and: 15  ⟶  true

nums := Array newFrom: (1 to: 5).
nums at: 1 put: 6.
nums  ⟶  #(6 2 3 4 5)
```

> 키워드 메시지는 한 개 이상의 인자를 요구합니다. 그 인자들의 선택자는 각각 콜론 (:) 으로 끝나는 한 개 또는 그 이상의 키워드들로 구성됩니다. 키워드 메시지는 다음의 문법구조로 되어있습니다:
> receiver `selectorWordOne:` argument-One `wordTwo:` argumentTwo

4.3 메시지 구성하기

3종류의 메시지는, 명쾌한 방식으로 메시지가 작성될 수 있도록 해주는 각각 다른 우선순위를 가지고 있습니다.

1. 단항 메시지는 항상 먼저 전달되며, 그 다음 바이너리 메시지이고, 마지막으로 키워드 메시지가 보내집니다.

2. 괄호 안에 있는 메시지는 어떤 종류의 메시지 보다 먼저 보내집니다.

3. 동일한 종류의 메시지는 왼쪽에서부터 오른쪽으로 계산됩니다.

이 규칙들은 매우 자연스러운 읽기 순서로 이어집니다. 지금, 만약 당신이 원하는 순서대로 메시지가 전달되기를 원한다면, 그림 4.3 에서 볼수있듯이 원하는만큼 괄호들을 추가로 사용하면 됩니다. 이 그림에서, 메시지 yellow는 단항 메시지이며, 메시지 color:는 키워드 메시지이므로, 우선순위에 의해 메시지 전송식 Color yellow 가 먼저 전달됩니다. 하지만 굳이 Color yellow 주위에 (필요없는) 괄호를 넣은 것은 단지 이 메세지전송식이 먼저 보내진다는것을 강조하는 의미가 있습니다. 나머지 장에서는 이 강조할 점을 설명합니다.

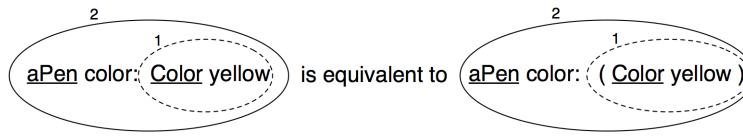


그림 4.3: `Color yellow`가 전달되어야 하기때문에 단항메시지(`yellow`)가 먼저 전송됩니다. 이 단항 메세지는 `aPen color:`의 메세지로 전달될 `color` 객체를 반환합니다..

단항 > 이항 > 키워드

단항 메시지를 먼저보내고, 그 다음이 바이너리 메시지이며, 마지막으로 키워드 메시지가 전달됩니다. 또한 다른 종류의 메시지보다 단항 메시지가 더 높은 우선순위를 가지게 됩니다.

> **Rule One.** 단항 메시지가 먼저 전달되고, 그 다음이 바이너리 메시지이며, 마지막으로 키워드 메시지가 전달됩니다.
>
> `Unary > Binary > Keyword`

아래의 예제에서 볼 수 있듯이, 스몰토크의 문법 규칙은 일반적으로 보내는 메시지가 자연스럽게 보여질 수 있도록 합니다:

```
1000 factorial / 999 factorial  ⟶  1000
2 raisedTo: 1 + 3 factorial     ⟶  128
```

안타깝게도, 보내지는 산술메시지들은 단순하게 처리되기 때문에, 바이너리 연산자들에 대한 우선순위를 원할 때마다 괄호를 넣는것이 좋습니다.

```
1 + 2 * 3    ⟶  9
1 + (2 * 3)  ⟶  7
```

약간 더 복잡한(!) 다음 예는 심지어 복잡한 스몰토크표현식도 자연스럽게 보여질 수 있다는 것을 훌륭하게 보여줍니다:

```
[:aClass | aClass methodDict keys select: [:aMethod |
    (aClass>>aMethod) isAbstract ]]
  value: \ct{Boolean}  ⟶  an IdentitySet(#or: #| #and: #&
    #ifTrue: #ifTrue:ifFalse: #ifFalse: #not #ifFalse:ifTrue:)
```

이제 우리가 알아보려는것은 어떤 `Boolean`클래스의 메서드가 추상요소인가 하는 것입니다[3]. 메서드 사전*dictionary* 의 키를 위해, aClass 라는 인자클래스를 이 클래스의 추상메서드를 선택합니다. 그 다음, `Boolean`이라는 실체적인 값 에 인자 `aClass`를(arugument aClass)를 바인딩합니다. 단항 메시지 `isAbstract`를 메서드에 보내기 전에, 클래스에서 메서드를 선택하는 `>>` 이항 메시지를 보내는 용도로만 쓰일 괄호가 필요합니다. 결과적으로 `Boolean`의 실체적 하위클래스인 `True`와 `False`로 구현한 메서드를 볼 수 있습니다.

Example. 메시지 `aPen color: Color yellow` 메시지는, 클래스 Color로 보내지는 단항메세지와 aPen으로 보내지는 Color: 라는 키워드 메시지를 가지고 있습니다. 예 4.1을 보면 우선순위에 의해 단항 메시지가 먼저 전달되기 때문에 `Color yellow`가 전달됩니다(1번). 이 메시지 전달식은 보신바와 같이 메시지 `aPen color: aColor` 의 인자로서 전달되는 color 객체입니다(2번). 그림 4.3 은 어떻게 메시지가 발송되는지 그림으로 보여줍니다.

[3]사실, 비슷한 수준으로 작성할 수 있지만 이지만 더 간단한 표현을 쓰겠습니다: `Boolean methodDict select: #isAbstract thenCollect: #selector`

Example 4.1: `aPen color: Color yellow`의 처리 분석하기

```
    aPen color: Color yellow
(1) Color yellow          "단항 메시지가 먼저 전덜되고"
       ⟶  aColor
(2) aPen color: aColor    "그 다음 키워드 메시지가 전달됩니다"
```

Example. 메시지 `aPen go:100 + 20`은, 바이너리 메시지 `+ 20`과 키워드 메시지 `go:`를 가지고 있습니다. 바이너리 메시지는 키워드 메시지 전에 전달되므로, `100 + 20`이 먼저 전달됩니다. (1번): 메시지 `+ 20`은 100 이라는 객체로 전달되며, 숫자 120을 리턴합니다. 그 다음, 메시지 `aPen go: 120`에서, 120은 `go:` 셀렉터의 인자로서 함께 전달됩니다(2번). 예제 4.2 는 메시지가 어떻게 실행되는지를 보여줍니다.

Example 4.2: `aPen go: 100 + 20` 분해하기

```
    aPen go: 100 + 20
(1) 100 + 20           "이항 메시지가 먼저 전달되고"
      --> 120
(2) aPen go: 120       "그 다음 키워드 메시지가 전달됩니다"
```

Example. 연습으로 하나의 단항 메시지와 하나의 키워드, 하나의 이항 메시지(그림 4.5 참조)로 구성된 Pen new go: 100 + 20 의 처리과정을 분석해 보겠습니다.

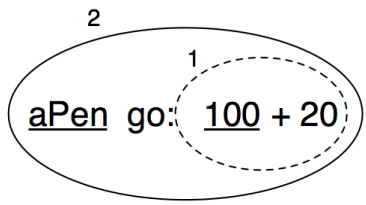


그림 4.4: 이항 메시지가 키워드 메시지보다 우선적으로 전달 되었습니다.

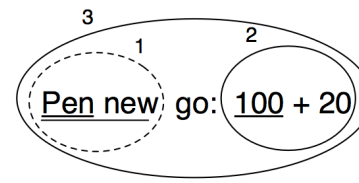


그림 4.5: `Pen new go: 100 + 20` 구문의 분해

괄호 우선

Rule Two. 괄호로 묶인 메시지는 다른 메시지 보다 먼저 전달됩니다.

(메시지) > 단항 메세지 > 이항 메세지 > 키워드메세지

```
1.5 tan rounded asString = (((1.5 tan) rounded) asString)  ⟶
    true
3 + 4 factorial    ⟶  27    "(5040이 아닙니다)"
(3 + 4) factorial  ⟶  5040
```

여기서 play이전에 lowMajorScaleOn:을 강제로 보내려면 괄호가 필요합니다.

```
(FMSound lowMajorScaleOn: FMSound clarinet) play
"(1) 클라리넷 소리를 만들기 위해 clarinet 메시지를 FMSound 클래스에
     보냅니다.
 (2) FMSound에 인자로 보낸 소리를 lowMajorScaleOn: 키워드 메시지로
     보냅니다.
 (3) 결과 소리를 재생합니다."
```

Example. (65@325 extent: 134 @ 100) center라는 메시지는 상단 왼쪽 꼭지점이 $(65, 325)$ 이며 그 크기가 134×100 인 직사각형의 중심을 반환합니다. 예제 4.3 은 어떻게 메시지가 분해되며 전달되는지를 보여줍니다. 먼저 괄호 사이에 있는 메시지를 보냅니다. 여기에는 먼저 보내어 포인트 정보를 돌려 받을 65@325(1번)와 134X100(2번) 두 개의 이항 메시지가 있으며, 그 다음에 134×100 의 정보를 보내고 사각형 정보를 돌려 받을 extent:(3번) 키워드 메시지가 있습니다. 마지막으로 단항 메시지 center는 직사각형에 보내지며 (4번), 점 정보를 돌려 받습니다. 100 이라는 객체는 center라는 메세지를 이해하지 못하기때문에 괄호없이 메시지를 처리하면 오류가 발생합니다

Example 4.3: *Example of Parentheses.*

```
    (65 @ 325 extent: 134 @ 100) center
(1) 65@325                              "binary"
     ⟶  aPoint
(2) 134@100                             "binary"
```

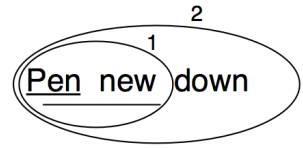


그림 4.6: Pen new down 분해

```
     ⟶  anotherPoint
(3) aPoint extent: anotherPoint          "keyword"
     ⟶  aRectangle
(4) aRectangle center                    "unary"
     ⟶  132@375
```

왼쪽에서 오른쪽으로

이제 우리는 다양한 종류의 메시지와 우선권이 취급되는 방식을 알게 되었습니다. 더 생각해봐야할 마지막 질문은, 동일한 우선권을 가진 메시지가 있는 경우 어떤 순서로 전달되는지에 대한것입니다. 동일한 우선권을 가진 메시지들은 왼쪽에서 오른쪽 순서로 전달됩니다. 예 4.3 에서 두 개의 점 정보 생성 메시지 (@)가 다른 것보다 먼저 전달되었던 것을 보셨던 것을 기억해주세요.

> **Rule Three.** 동일한 종류의 메시지가 있을 때, 처리 순서는 왼쪽에서 오른쪽 입니다.

Example. 보낼메시지 `Pen new down` 에서 모든 메시지는 단항 메시지이므로, 가장 왼쪽의 `Pen new`가 가장 먼저 처리됩니다. 이것은 그림 4.6 에서 볼수 있듯이, 두 번째 메시지인 down이 새롭게 만들어진 Pen 객체를 리턴합니다.

산술적 비일관성

메시지 작성 규칙들은 단순하지만, 바이너리 메시지로 표현된 산술 메시지를 실행하는경우 비일관성이 생기게 됩니다. 이제부터 괄호가 필요한 일반적인 상황들에 대해 알아볼 것입니다.

```
3 + 4 * 5       ⟶  35    "(not 23)"
3 + (4 * 5)     ⟶  23
1 + 1/3         ⟶  (2/3) "and not 4/3"
1 + (1/3)       ⟶  (4/3)
1/3 + 2/3       ⟶  (7/9) "and not 1"
(1/3) + (2/3)   ⟶  1
```

Example. 메시지 보내기 `20+2*5`에서, 바이너리 메시지는 `+`와 `*`뿐 입니다. 그럼에도 불구하고, 스몰토크에서는 연산 `+`와 `*`를 위한 특정 우선권은 존재하지 않습니다. 이것들은 단지 바이너리 메시지 이므로, `*`는 `+`에 대해 우선권을 갖지 않습니다. 예제 4.4 에서 볼 수 있듯이, 이 경우 제일 왼쪽의 `+`가 전달되며 (1번) 그 다음 `*`가 해당결과에 전달됩니다 (2번).

Example 4.4: `20 + 2 * 5` 분해하기

```
"바이너리 메시지들 중에서 우선순위는 없기때문에,
 산술규칙 *가 첫 번째로 전달되어야 하지만 가장
 왼쪽의 메시지인 +가 첫 번째로 처리됩니다"
     20 + 2 * 5
(1)  20 + 2  ⟶  22
(2)  22 * 5  ⟶  110
```

예제 4.4 에서 볼 수 있듯이, 이 메시지 전송 결과는 30이 아니며 110입니다. 생각했던 결과는 아니지만, 메시지를 보내기 위해 사용된 규칙을 바로 따랐을 것입니다. 결국 스몰토크모델의 단순성때문에 이런 결과가 나오게 됩니다. 원하는 올바른 결과를 얻으려면 괄호를 사용해야 합니다. 메시지를 괄호로 둘러쌓으면 괄호로 둘러싼 메시지를 먼저 처리합니다. 따라서 보내는 메시지 20 + (2 * 5) 가 반환하는 결과는 예제 4.5 에서 확인할 수 있습니다.

Example 4.5: `20+(2*5)` 분해하기

```
"괄호로 둘러 쌓인 메시지를 먼저 처리하므로 *를 + 보다 우선순위를
 주어 올바른 처리를 유도합니다."
    20 + (2 * 5)
(1) (2 * 5)  ⟶  10
(2) 20 + 10  ⟶  30
```

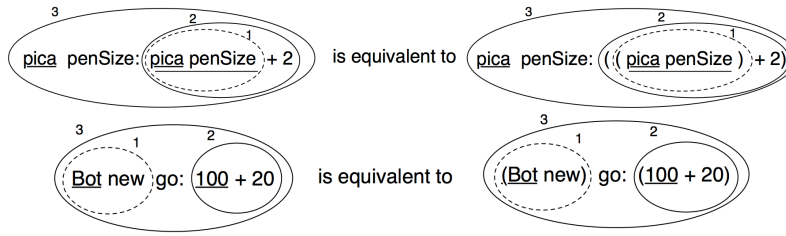


그림 4.7: 괄호를 사용하는 등가 메시지

Implicit precedence	Explicitly parenthesized equivalent
aPen color: Color yellow	aPen color: (Color yellow)
aPen go: 100 + 20	aPen go: (100 + 20)
aPen penSize: aPen penSize + 2	aPen penSize: ((aPen penSize) + 2)
2 factorial + 4	(2 factorial) + 4

그림 4.8: 메시지 보내기와 괄호를 넣은 등가물

> 스몰토크에서는 연산 +와 * 와 같은 산술 연산자는 다른 우선권을 갖지 않습니다. +와 *는 단지 바이너리 메시지일 뿐이므로, *는 +에 대해 우선권을 갖지 않습니다. 원하는 결과를 얻기 위해 괄호를 사용합니다.

참고로 첫 번째 규칙에서 바이너리 메시지와 키워드 메시지 이전에 발송된 단항 메시지는 명확한 구문을 위해 괄호 넣기를 하지 않는다라는 규칙을 기억해주세요. 표 4.8은 규칙을 따라 작성한 메시지전송식과 규칙이 존재하지 않을 경우 의미가 같은 메시지전송식을 보여주고 있습니다. 두 메시지는 같은 결과를 내며 같은 결과를 반환합니다.

4.4 키워드 메시지를 식별하기 위한 힌트

종종, 초보자들은 언제 괄호를 넣어야 할지 판단하는걸 어려워합니다. 이제, 컴파일러가 어떻게 키워드 메시지를 인식하는지 살펴보겠습니다.

괄호를 넣을지에 대한 여부

[,], 문자는 범위를 뚜렷하게 구분해줍니다. 구분된 각 범위 내에서, 문자 ., 또는 ; 로 잘리지 않고 :(콜론)으로 끝나는 가장 긴 단어 시퀀스를 키워드 메세지라고 합니다. [,], 문자가 :(콜론)이 붙은 단어를 둘러 쌓았다면, 이 단어는 해당 영역 한정으로 정의된 키워드 메시지가 됩니다.

아래의 예에는, 두 가지의 명백한 키워드 메시지가 있습니다. `rotatedBy:magnify:smoothing:` 과 `at:put:`. 입니다.

```
aDict
   at: (rotatingForm
          rotateBy: angle
          magnify: 2
          smoothing: 1)
   put: 3
```

> [,], (,) 문자로 명확하게 범위를 정한 영역. 이렇게 구분된 각 범위 내에서, 문자 . 또는 ; 로 잘리지 않고 :(콜론)으로 끝나는 가장 긴 단어 시퀀스를 키워드 메세지라고 합니다. [,], 문자가 :(콜론)이 붙은 단어를 둘러 쌓았다면, 이 단어는 해당 영역 한정으로 정의된 키워드 메시지가 됩니다.

HINT 만약 이러한 우선순위 규칙이 어렵다면, 동일한 우선순위를 가진 두 개의 메시지를 구별하고 싶을때마다 사용자가 스스로 언제든지 괄호를 사용하면 됩니다.

메시지 전송식 `x isNil`은 단항메시지 이므로, 키워드 메시지 `ifTrue:`보다 먼저 전달되기 때문에 아래 코드는 괄호가 필요 없습니다.

```
(x isNil)
   ifTrue:[...]
```

아래 코드에서 메시지 includes:와 ifTrue:는 둘 다 키워드 메시지이기때문에 괄호를 필요로 합니다.

```
ord := OrderedCollection new.
(ord includes: $a)
   ifTrue:[...]
```

괄호가 없다면 `includes:ifTrue:` 라는 정의되지 않은 메시지는 곧바로 ord 라는 컬렉션으로 보내지게 됩니다.

언제 [] 또는 ()를 사용할까요?

당신은 괄호보다 꺽쇠괄호를 사용할 때를 더 어렵게 느낄수도 있습니다. 0순위로 구문을 처리하고싶은경우 [ ] 를 사용하면 되고, 이것이 바로 [ ] 괄호 사용의 기본 원칙입니다. [expression] 구문으로 상황에 따라 반복처리(0번도 가능)가 가능한 블록(즉 블록객체)을 만들 수 있습니다. 참고로 안에 들어갈 구문은 메시지전송식, 변수, 리터럴, 할당문 또는 어떤 것이든 가능합니다.

이런 이유때문에, ifTrue: 또는 ifTrue:ifFalse:의 상태분기문에서는 블록을 요구합니다. 같은 이유로 수신자와 whileTrue: 메시지의 인자는 이것(수신자와 인자)들의 반복처리 횟수를 모르는경우 꺽쇠 괄호를 사용해야 합니다.

꺽쇠괄호와 달리, 괄호는 메시지를 보내는 순서에만 영향을 미칩니다. 따라서 (expression)의 상황에서 expression은 반복없이 한번만 처리됩니다.

```
[ x isReady ] whileTrue: [ y doSomething ] "수신자와 인자
                                    둘 다 블록이어야 합니다"
4 timesRepeat: [ Beeper beep ] "인자를 반복적으로 처리해야
                                하기 때문에 블록이어야 합니다"
(x isReady) ifTrue: [ y doSomething ] "수신자(이경우 ifTrue:)는
                          한번만 처리되므로, 블록이 필요 없습니다"
```

4.5 프로그램식의 처리 순서

구두점으로 분리되는 프로그램식(즉 메세지전송, 할당)등은 순서대로 처리됩니다. 참고로 변수 정의와 뒤따라오는 프로그램식 사이에는 구두점이 없습니다. 프로그램식의 실행을 진행해서 얻어지는 값은 처리되는 프로그램식들중 가장 마지막 구문의 값이 됩니다. 여러 프로그램식을 한꺼번에 진행할때 모든

프로그램식의 반환값은 마지막 한개를 제외하고 모두 무시됩니다. 구두점은 구분자이며 프로그램식의 종결자(terminator)[4]가 아닌것에 주의해주세요. 따라서 마지막 구두점은 선택적으로 사용하면 됩니다.

```
| box |
box := 20@30 corner: 60@90.
box containsPoint: 40@50  ⟶  true
```

4.6 캐스케이드된 메시지

스몰토크는 세미콜론 (;)을 사용하여 동일한 수신자에게 여러개의 메시지를 보낼 수 있는 방법이 있습니다. 이것을 스몰토크용어로 캐스케이드라고 합니다.

표현식 Msg1 ; Msg2

```
Transcript show:
'Squeak is '.
Transcript show:
'fun '.
Transcript cr.
```

is equivalent to:

```
Transcript show:
'Squeak is'; show:
'fun '; cr
```

캐스케이드된 메시지를 받는 객체는 단순히 객체만 되어야 하는것이 아니라 스스로에게 메시지를 보낸 결과 자체가 대상이 될 수도 있습니다(아래의 경우는 instanse). 결국 모든 캐스케이드된 메시지의 수신자는 캐스케이드에 걸려있는 요소들중 첫번째 메시지를 수신하는 객체(또는 인스턴스)가 됩니다. 다음 예제에서 첫째로 캐스케이드된 메시지인 setX:setY[5]는 Point new의 처리결과로 만들어진 Point이며 Point 자체가 대상이 되는건 아닙니다. 그 다음 메시지 isZero는 동일한 수신자에게 전달되는것을 볼 수 있습니다.

[4] C의 경우는 ;(세미콜론)이 현재 문장의 종결자이기때문에 이런 표현이 쓰인것같습니다.

[5] Point new 자체를 인스턴스를 만들기 위한 하나의 단위로 본다면 setX:setY는 첫번때 캐스케이드 메시지기 맞습니다.

```
Point new setX: 25 setY: 35; isZero  ⟶  false
```

4.7 4장 요약

- 메시지는, 다른 메시지 전송의 결과가 될 수 있는 수신자인 객체로 전달됩니다.
- 단항 메시지는 어떤 인수도 필요하지 않은 메시지 입니다. 단항 메시지들은 receiver `selector`의 형태를 갖고 있습니다.
- 이항 메시지는–수신자와 다른 객체–두 개의 객체에 관여하는 메시지이며, 이항 메시지의 선택자*selector*는 다음 목록의 기호 : +, -, *, /, |, &, =, >, <, ~과 @의 한 개 또는 그 이상의 것으로 작성합니다. 이항 메시지들은 receiver `selector` argument의 형태를 갖고 있습니다.
- 키워드 메시지는 하나이상의 객체를 가지는 메시지이며, 적어도 한 개의 콜론 문자(:)를 갖고 있습니다.
  이 키워드 메시지들은 receiver `selectorWordOne:` argumentOne `wordTwo:` argumentTwo의 형태를 갖고 있습니다.
- **Rule One.** 단항 메시지는 제일 먼저 발송되며, 그 다음 바이너리 메시지, 마지막으로 키워드 메시지가 전달됩니다.
- **Rule Two.** 괄호로 묶인 메시지는 다른 메시지 보다 먼저 전달됩니다.
- **Rule Three.** 동일한 종류의 메시지가 있을 때, 처리 순서는 왼쪽에서 오른쪽 입니다.
- 스몰토크에서, +와 *와 같은 전통적인 산술 연산자는 동일한 우선순위를 가집니다. +와 *는 이항 메시지 이므로, *는 +에 대해 우선순위를 갖지 않습니다. 다른 결과를 얻으려면 반드시 괄호를 사용하셔야 합니다.

제 II 편

스퀵 개발

제 5 장

스몰토크객체 모델

스몰토크의 프로그래밍 모델은 단순하며 일관적입니다: 모든 것은 객체이며, 객체는 서로 메시지를 보내어 의사소통을 합니다. 하지만, 이러한 단순성과 일관성은 다른 언어를 사용한 프로그래머에게 어려움을 느끼게 하는 요인이 될 수 있습니다. 이 장에서는 스몰토크객체 모델의 핵심 개념을 보여드리고, 클래스와 객체 표현의 중요성에 대해 이야기 하도록 하겠습니다.

5.1 스몰토크의 모델 규칙

스몰토크객체 모델은 일관성있게 적용한 간단한 규칙 모음을 기반으로 합니다. 규칙은 다음과 같습니다:

Rule 1. 모든 요소는 객체입니다.

Rule 2. 모든 객체는 클래스의 인스턴스입니다.

Rule 3. 모든 클래스는 super 클래스가 있습니다.

Rule 4. 모든 동작은 메시지를 보낼 때 일어납니다.

Rule 5. 메서드 탐색은 상속 관계를 따릅니다.

이들 규칙에 대해 좀 더 자세히 알아보도록 하겠습니다.

5.2 모든 요소는 객체입니다

"모든 요소는 객체입니다" 라는 문구는 상당한 중독성이 있습니다. 스몰토크로 잠깐 동안 일을 해본다면, 하고 있는 모든 일을 어떻게 단순화 하는지 놀라기 시작할 것입니다. 예를 들어, Integer는 실제로 객체이므로, 여러분은 다른 객체에 메시지를 보냈던 것처럼, 정수에도 역시 메시지를 보낼 수 있습니다.

```
3 + 4         ⟶ 7 "7을 얻기 위해 + 를4' 으로3 보냅니다'"
20 factorial ⟶ 2432902008176640000 "큰 수를 얻기 위해
                                    팩토리얼을 보냅니다""
```

`20 factorial`이라는 표현은 `7`이라는 표현과는 분명히 다르지만, 둘 다 객체이므로, 어떤 코드도 — 심지어 `factorial`의 구현까지도 — 알 필요가 없습니다. 아마도 이 규칙의 근본적인 결론은 다음과 같을지도 모릅니다.

> 클래스 또한 객체입니다.

더 나아가, 클래스는 2등급 객체가 아닙니다: 메시지를 보내고, 점검하는 등을 할 수 있는 진정한 1등급 객체입니다. 이는 스퀵이야말로 개발자에게 다양한 표현 가능성을 제공하는 진정한 반응 시스템임을 의미합니다.

스몰토크의 구현 내용을 깊이 들어가보면, 세 가지 종류의 오브젝트가 있습니다. (1) 참조로 전달하는 인스턴스 변수인 일반 객체, (2) 값을 전달하는 크기가 작은 정수, (3) 메모리의 연속된 공간을 유지하는 배열과 유사한 색인 가능 객체가 있습니다. 스몰토크의 아름다움은 이들 세가지 객체의 차이에 대해 일반적으로 신경 쓸 필요가 없다는 것입니다.

5.3 모든 객체는 클래스의 인스턴스입니다

모든 객체는 클래스를 가지고 있으며, 스몰토크는 객체에 `class`라는 메시지를 보내어 원하는 객체의 클래스를 알아낼 수 있습니다.

```
1 class              ⟶  SmallInteger
20 factorial class   ⟶  LargePositiveInteger
'hello' class        ⟶  ByteString
#(1 2 3) class       ⟶  Array
(4@5) class          ⟶  Point
Object new class     ⟶  Object
```

클래스는 인스턴스 변수를 통해 클래스 스스로에 대한 인스턴스의 *구조*를 정의하고, 메서드를 통해 클래스 인스턴스의 *동작*을 정의합니다. 메서드는 다른 곳에서 호출될 수 있는 고유한 이름을 가진 셀렉터로서 클래스안에 있습니다. *클래스는 객체*이고, *모든 객체는 클래스의 인스턴스*이기 때문에, 클래스는 반드시 클래스의 인스턴스가 되어야 한다는 규칙이 따라옵니다. 클래스가 인스턴스인 클래스를 메타클래스 라고 합니다. 클래스를 만들 때마다 언제든지 시스템은 메타 클래스를 자동으로 만들어줍니다. 메타 클래스는 자신의 인스턴스가 되는 클래스의 구조와 동작을 정의합니다. 99%의 경우, 메타 클래스에 대해 생각할 필요가 없으며, 기꺼이 무시해도 됩니다. (12장 에서 메타클래스를 좀 더 자세히 살펴보겠습니다.)

인스턴스 변수

스몰토크에서 인스턴수 변수는 해당 인스턴스 내부(private)에서만 사용합니다. 스몰토크와는 반대로 인스턴스 변수에 대한 외부접근을 허용하는 Java와 C++등에서는(이러한 변수를 필드 또는 멤버변수라고 합니다) 같은 클래스에서 만들어진 다른 어떤 인스턴스에서도 인스턴스 변수에 접근이 가능합니다. Java와 C++에서는 객체의 캡슐화 범위를 클래스까지로 보지만 스몰-토크에서는 인스턴스까지를 캡슐화 범위로 생각해야 합니다.

스몰토크에서, 동일한 클래스로 만든 두 개의 인스턴스는, 클래스가 "접근자 메서드"를 정의하지 않으면, 서로 다른 인스턴스의 인스턴스 변수에 접근할 수 없습니다. 스퀵은 자신외의 그 어떤 다른 객체도 인스턴스 변수로 바로 접근하는 기능을 제공되는 문법을 언어에서 지원하지 않습니다. (사실 reflection(반영적)이라고 하는 기법은 메타 프로그래밍처럼 다른 객체에서 인스턴스 변수의 값을 요청할 수 있는 방법을 제공합니다: 메타 프로그래밍은

객체 인스펙터처럼 다른 객체 내부를 살펴볼 수 있는 도구를 위해 만들어졌습니다.)

인스턴스 변수는 인스턴스를 생성한 클래스에서 정의된 인스턴스의 메서드 또는 인스턴스를 정의한 클래스의 하위클래스에서 정의한 메서드 중 어떤 메서드를 사용해도 접근이 가능합니다. 이것은 스몰토크인스턴스 변수들이, C++과 Java에서, 보호(Protected)된 변수들과 비슷하다는 뜻이 됩니다. 그럼에도 불구하고, 스몰토크에서는 이 인스턴스 변수들을 개별적(private, 혹은 전용)이라고 부르는것을 선호하는데, 이유는 서브클래스에서부터 직접 인스턴스 변수에 접근하는 형식을, 좋지 않게 생각하기 때문입니다.

Example

`Method Point>>dist:`(메서드 5.1)는 수신자와 다른 점 사이의 거리를 계산합니다. 수신자의 인스턴스 변수인 x와 y 두개는 메서드 안에서 직접 접근이 가능합니다. 그러나 자신 외의 다른 점의 인스턴스 변수들은 aPoint로 들어온 객체의 x와 y 메서드에 메시지를 보내서 값을 받는 방법으로 변수에 접근해야 합니다.

Method 5.1: 두 점 사이의 거리

```
Point>>dist: aPoint
  ""
    | dx dy |
    dx := aPoint x -- x.
    dy :=  aPoint y -- y.
    ↑ ((dx * dx) + (dy * dy)) sqrt
```

클래스 기반 캡슐화 보다 인스턴스 기반 캡슐화를 선호하는 이유는, 인스턴스로 생성된 객체를 사용하려 할 때 추상화된 인스턴스의 사용법만 동일하다면 인스턴스 내부의 구현을 다르게 만들 수 있기 때문입니다. 예를 들어, `method point>>dist:` 메서드는 인자 aPoint가 수신자와 동일한 클래스의 인스턴스인지의 여부를 알 필요도 없고 신경쓸 필요도 없습니다. 인자 객체가 메시지 x와 y에 반응된다면, aPoint라는 객체는 끝 좌표가 될수도 있고, 데이터베이스에서 얻어진 데이터가 될 수도 있고 분산된 시스템의 다른컴퓨터의 데이터가

될 수도 있습니다. 메시지 x와 y에서 응답을 얻을 수 있을때까지, 메서드 5.1의 코드는 잘 작동합니다.

메서드

모든 메서드는 public 입니다[1]. 메서드들은 목적에 따라 프로토콜을 이용해서 분류됩니다. 몇가지 공통으로 쓰이는 프로토콜 이름은 관례에 따라 만들어 졌습니다. 모든 접근자 메서드에 대한 접근과, 객체에 적용하는 초기화 상태를 만들기 위한 initialization 프로토콜이 이런 관례의 예가 되겠습니다. private 프로토콜은 가끔 외부에서 보면 안되는 메서드를 그룹으로 만들 때 사용합니다. 그렇다고 해서 private메서드로 구현된 셀렉터에 메시지를 보내는것을 막는 방법은 없습니다.[2]

메서드는 객체의 모든 인스턴스 변수에 접근할 수 있습니다. 일부 스몰토크개발자들은 오직 접근자*accessor* 만 인스턴스 변수에 접근할 수 있도록 하는 방식을 좋아합니다. 이런 방식을 활용하는 경우는 있지만, 사용자의 클래스 인터페이스를 어수선하게 만들며, 외부에서 인스턴스 또는 클래스에 대한 private의 상태를 알 수 있는 가능성이 됩니다.

인스턴스 관점과 클래스 관점

클래스는 곧 객체이므로, 클래스는 인스턴스 변수와 메서드를 보유할 수 있습니다. 이런것들을 클래스 인스턴스 변수 와 클래스 메서드라고 하지만, 보통의 인스턴스 변수, 메서드와는 실제로 차이가 있는건 아닙니다: 클래스 인스턴스 변수들은 단지 메타클래스로 정의되는 인스턴스 변수들이며, 클래스 메서드는 단지 메타클래스에 의해 정의되는 메서드 일 뿐입니다.

클래스와 그 클래스의 메타클래스는 두 개의 구별된 클래스 이며, 심지어 클

[1] 글쎄요, 대부분, 스퀵에서, 문자열 `pvt` 로 시작하는 셀렉터인 메서드는 private입니다: `pvt` 메시지는 클래스(인스턴스) 내부에서만 사용할 수 있죠. 이런 특성이 있어도 `pvt` 메서드를 많이 사용하지는 않습니다.

[2] private 프로토콜로 메서드 또는 셀렉터를 만들었다고해서 그게 딱히 외부에서의 접근에 대해 보호받는건 아니라는 의미입니다. 분류는 해놓을 수 있지만 굳이 사용하려한다면 사용은 할 수 있다는 의미인거죠

래스는 메타클래스의 인스턴스이기도 합니다. 하지만, 이런 내용은 프로그래머에게 크게 상관은 없습니다: 당신은 객체와 클래스를 만드는것에 신경쓰면 됩니다.

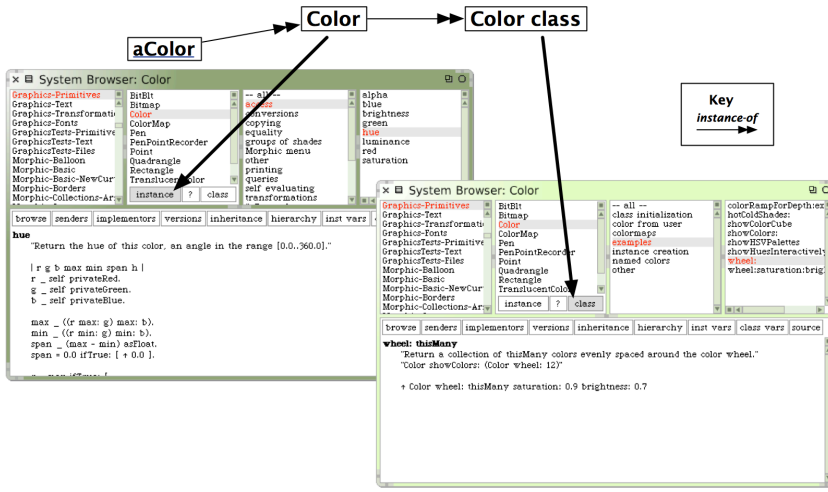


그림 5.1: 클래스와 그 자체의 메타 클래스(metaclass)를 검색하기

이렇게 클래스는 메타클래스의 인스턴스 라는 특성때문에, 그림 5.1 에 보이는 것처럼 System Browser에서는 클래스와 메타클래스를 "instance" side 그리고 "class" side의 양면성을 지닌 한몸처럼 탐색할 수 있습니다. 클래스 color 등을 검색하기 위해 instance 버튼을 클릭하면, blue와 같은 color의 인스턴스에 메시지를 보냈을때, 실행되는 메서드를 탐색하면 됩니다. class 버튼을 누르면, 클래스 Color class 를 탐색합니다. 이런 방법으로 당신은 color 클래스에 메시지가 실제로 보내질 때, 실행되는 메서드를 볼 수 있습니다. 예를 들어, Color blue는 클래스 color에 메시지를 보냅니다. 그렇게 되면, 인스턴스 관점이 아닌, color의 클래스 관점에서 정의된 메서드 blue 를 찾게됩니다.

```
aColor := Color blue.      "Class side method blue"
aColor       ⟶  Color blue
aColor red   ⟶  0.0        "Instance side accessor method red"
aColor blue  ⟶  1.0        "Instance side accessor method blue"
```

클래스는 System Browser의 인스턴스 관점에서 언급한 템플릿에 내용을 채워 정의합니다. 이렇게 내용이 채워진 템플릿을 Accept 하게되면, 시스템은 사용자가 정의한 클래스뿐만 아니라, 이에 대응하는 메타클래스도 같이 만들게 됩니다. 또한 System Browser의 class 버튼을 클릭하여 메타클래스를 검색하는것도 가능합니다. 메타클래스 생성 탬플릿에서 인스턴스 변수이름 목록 외에 당신이 편집할 수 있는건 없습니다.

클래스를 만들고 나면, instance 버튼을 클릭하면, 클래스의 인스턴스(와 클래스의 서브클래스)가 소유할 메서드를 편집하고 탐색할 수 있습니다. 이런 방법으로, 그림 5.1 처럼 클래스 color의 인스턴스에서 정의된 메서드 hue를 확인할 수 있습니다. 반대로, class 버튼을 이용하면 메타클래스(이번 경우 Color 클래스)를 검색하고 편집할 수 있습니다.

클래스 메서드

클래스 메서드는 그럭저럭 쓸모가 있습니다. 좋은 예 몇가지를 들어보기 위해 Color class를 탐색하겠습니다. 클래스에 정의한 메서드에는 두 가지 종류가 있음을 발견하실 것입니다: `Color class>>blue`와 같이 클래스의 인스턴스를 만드는 메서드들과 `Color class>>showColorCube` 처럼, 유틸리티 함수를 수행하는 메서드가 있죠. 가끔 다른 방식으로 사용하는 클래스 메서드를 보겠지만, 이 두 가지 종류가 일반적입니다.

클래스 측면(the class side)에서 유틸리티 메서드들을 배치하는 작업은 편리하며, class side에서 유틸리티 메서드를 배치하기 편리한 이유는, 추가 객체를 먼저 생성할 필요 없이 메서스들을 실행해볼 수 있기 때문입니다. 이런 메서드 대부분은 쉽게 실행할 수 있도록 설계시 주석이 들어가 있습니다.

 메서드 `Color class>>showColorCube`를 검색하고, 주석 ``Color show ColorCube'' 에서 따옴표 내부를 더블 클릭한 다음, *CMD−d* 를 타이핑 합니

다.

이 메서드가 실행되는 결과를 확인할 수 있습니다. 실행결과를 취소(undo)하기 위해
World ▷ restore display (r) 을 선택합니다.

Java 와 C++ 에 익숙하다면, 클래스 메서드가 정적 메서드(static method)와 비슷하게 보일겁니다. 하지만, 스몰토크에서는 그렇지 않습니다: 비록 Java의 메서드가 정적인 프로시져*statically resolved procedures* 라고해도, 스몰토크클래스 메서드는 동적으로 내보내는 $\overset{textrmdynamically}{\text{dispatched}}$ 메서드입니다. 이런 특성은 스몰토크에서는 클래스의 메서드에 대해 동적으로 상속*inheritance*, 재지정*overriding*, super-send등이 가능하지만 Java의 정적메서드에서는 이런 동작을 처리하지 못함을 의미합니다.

클래스 인스턴스 변수

일반적인 인스턴스 변수처럼, 클래스 인스턴스 변수는 클래스의 모든 인스턴스같이 동일한 이름의 변수를 가지게 되고, 현 클래스의 하위클래스에 대한 인스턴스는 이러한 변수 이름들을 상속받습니다. 다만 개별의 인스턴스는 각자의 값을 따로 가지게 되죠. 이런 성질은 클래스 인스턴스 변수들에게도 그대로 적용됩니다: 각 클래스는 그 자체로 private 클래스 인스턴스 변수를 가지게 됩니다. 하위클래스는 인스턴스 변수들이 가진 클래스 인스턴스 변수들을 상속하겠지만, 클래스 인스턴스 변수들의 고유사본을 갖게 됩니다. 객체가 인스턴스 변수들을 공유하지 않듯이, 클래스와 서브클래스간에는 클래스 인스턴스 변수를 공유하지 않습니다.

count라고 불리는 클래스 인스턴스 변수는 클래스에 따른 인스턴스가 얼마나 많은지를 알아보는데 사용할 수 있습니다. 하지만 서브클래스는 자신만의 count변수를 가지기때문에 하위클래스의 인스턴스는 별도의 인스턴스 카운트값을 가지게 됩니다.

Example: 클래스 인스턴스 변수는 서브클래스와 공유되지 않습니다. Dog(개)와 Dog(개)로부터 count 클래스 인스턴스 변수를 상속받는 Hyena(하이에

나) 클래스를 정의한다고 가정해보겠습니다.

Class 5.2: *Dog*와 *Hyena*

```
Object subclass: #Dog
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'SBE--CIV'

Dog class
    instanceVariableNames: 'count'

Dog subclass: #Hyena
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'SBE--CIV'
```

이제 Dog클래스의 count(카운트)를 0으로 초기화 하는 Dog의 클래스 메서드를 정의하고, count는 새로운 인스턴스가 만들어질 때마다 하나씩 증가한다고 생각해보겠습니다:

Method 5.3: 새로운 *Dog*에 대한 갯수 유지

```
Dog class>>initialize
    super initialize.
    count := 0.

Dog class>>new
    count := count +1.
    ↑ super new

Dog class>>count
    ↑ count
```

지금, 새로운 Dog의 인스턴스가 만들어질때마다, Dog의 count가 증가되며, Hyena를 만들 때도 값은 증가하지만, Dog과 Hyena의 count는 따로 계산됩니다:

```
Dog initialize.
Hyena initialize.
Dog count    ⟶  0
Hyena count  ⟶  0
```

```
Dog new.
Dog count    ⟶  1
Dog new.
Dog count    ⟶  2
Hyena new.
Hyena count  ⟶  1
```

클래스 인스턴스 변수는 인스턴스 변수가 인스턴스에 대해 private 으로 처리되는 것처럼, 클래스 인스턴스 변수 또한 클래스에 자체적으로만 동작한다는 것을 기억해주세요. 클래스와 그 클래스의 인스턴스들이 전혀 다른 객체로 처리하는 등의 특성때문에 다음과 같은 결론을 얻을 수 있습니다.

> 클래스는 그 자체의 인스턴스 변수에 대한 접근성을 가지고 있지 않습니다.

> 클래스의 인스턴스는 그 자체의 클래스가 갖는 클래스 인스턴스 변수에 대한 접근 권한이 없습니다.

이러한 이유 때문에, 인스턴스 초기화 메서드는 항상 instance side에서 정의해야 합니다: class side에서는 인스턴스 변수에 대한 접근을 할 수 없기 때문에, 해당되는 변수들을 초기화할 수 없습니다. class side은 새롭게 만들어진 인스턴스에 접근자를 사용해서 초기화 메시지를 보내는 것 외에는 할 수 없습니다.

비슷한 이유로, 인스턴스는 이들 클래스에 대해 접근자 메시지를 사용해서 간접적으로 클래스 인스턴스 변수에 접근해야 합니다.

Java 는 클래스 인스턴스 변수와 비슷한 것이 없습니다. Java와 C++ 의 정적 변수는 **??**절 에서 알아볼 스몰토크의 클래스 변수에 더 가깝습니다.

Example: Singleton 을 정의합니다. Singleton 패턴 [3]은 클래스 인스턴스 변수와 클래스 메서드의 표준적인 사용방법입니다. WebServer 클래스를 구현하고 이 클래스에 대한 단 하나만의 인스턴스만 존재할 수 있도록

Singletone 패턴을 적용한다고 가정하겠습니다. 우리는 시스템 브라우저에서 instance 버튼을 클릭한 뒤, 아래처럼 (class 5.4) 클래스 WebServer를 정의할 수 있습니다.

Class 5.4: *singleton* 클래스

```
Object subclass: #WebServer
    instanceVariableNames: 'sessions'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Web'
```

그 다음, class 버튼을 클릭하여, 클래스 측면에서 인스턴수 변수 `uniqueInstance`를 추가합니다.

Class 5.5: *singleton* 클래스의 *class side*

```
WebServer class
    instanceVariableNames: 'uniqueInstance'
```

결과를 보면 WebServer 클래스는 superclass와 methodDict와 같은 상속 변수들뿐만 아니라 다른 인스턴스 변수도 갖고 있다는 것을 알 수 있습니다.

메서드 5.6 처럼 `uniqueInstance`라는 인스턴스 변수와 같은 이름으로 클래스 메서드메서드를 정의할 수 있습니다. 이 메서드는 `uniquenInstance` 인스턴스 변수를 초기화 했는지 확인합니다. 만약 인스턴스 변수를 초기화 하지 않았다면, 인스턴스 메서드를 만들고 클래스 인스턴스 변수 uniquenInstance에 만든 메서드를 할당할 것입니다. 마지막으로 uniquenInstance 값을 반환합니다. uniquenInstance는 클래스 인스턴스 변수이므로, 이 메서드는 그 클래스 인스턴스 변수에 직접 접근할 수 있습니다.

Method 5.6: *uniqueInstance (in class side)*

```
WebServer class>>uniqueInstance
     uniqueInstance ifNil: [uniqueInstance := self new].
     ↑ uniqueInstance
```

`WebServer uniquenInstance`를 실행한 뒤 제일 처음으로. `WebServer` 클래스의 인스턴스가 `uniqueInstance` 변수를 만들고 할당합니다. 그 뒤 이전에 만든 인스턴스를 새로 만든 인스턴스 대신 반환하게 되죠.

참고로 메서드 5.6 에 적혀있는 인스턴스 생성 코드는 WebServer new가 아닌 self new 처럼 작성했습니다. 이런 방법에는 어떤 차이가 있을까요? 당신은 uniquenInstance 메서드는 WebServer에 정의한 것이기 때문에, 이 둘은 동일한 수준에 있다 생각할 수 있습니다. 물론! WebServer클래스의 서브클래스를 만들어 내기 전까지는 같습니다. 하지만 ReliableWebServer 클래스가 WebServer의 서브클래스 이고 uniqueInstance를 상속한다고 가정한다면, ReliableWebServer uniqueInstance는 WebServer가 아닌 ReliableWebServer 클래스에 반응합니다 : 이 경우 self는 ReliableWebServer만을 대상으로 동작하게 하기 위한 확실한 방법이 됩니다. 왜냐하면 이런 (self를 사용하는 등의) 작업의 결과는 동작을 해당 클래스로 한정되게 해주기 때문이죠. 또한 WebServer와 RealiableWebServer가 uniqueInstance라고 하는 각각 별도의 클래스 인스턴스 변수를 가지게 된다는걸 기억하세요. 물론 이 두 개의 클래스의 인스턴스 변수는 제각기 다른 값을 가집니다.

5.4 모든 클래스는 super클래스를 가집니다

스몰토크에서 각각의 클래스는 유일한 하나의 super클래스로부터, 클래스의 동작 그리고 구조에 대한 설명을 상속받습니다. 스몰토크는 단일 상속을 지원합니다.

```
SmallInteger superclass  ⟶  Integer
Integer superclass       ⟶  Number
Number superclass        ⟶  Magnitude
Magnitude superclass     ⟶  Object
Object superclass        ⟶  ProtoObject
ProtoObject superclass   ⟶  nil
```

전통적으로, 스몰토크상속 계층에서 최상위는 Object 클래스 입니다 (모든 것이 객체이기 때문입니다). 스퀵에서, 최상위 요소는 실제로 ProtoObject라고 불리는 클래스가 됩니다만, 일반적인 경우 당신이 이 클래스에 신경을 쓸 필요는 없습니다. ProtoObject는 모든 객체가 반드시 가져야 할 메시지의 최소한의 요소를 캡슐화 하고 있습니다. 그렇지만 대부분의 클래스는 ProtoObject가 아닌 Object에게서 상속을 받으며, Object에는 ProtoObject에 비해 대부

분의 객체가 반드시 이해하고 응답해야 할 수많은 추가적인 메시지등이 정의되어 있습니다. 다른 방법을 써야 할 특별한 이유가 없다면, 프로그래밍으로 클래스를 만들때, 만들어지는 클래스는 Object 또는 Object의 서브클래스 중 하나가 됩니다.

 새로운 클래스는 일반적으로 `subclass:instanceVariableNames:...`이라는 메시지를 보냄으로써 만들어집니다. 클래스를 만들기 위한 몇 가지 다른 메서드가 있습니다. 그 메서드들이 무엇인지는 프로토콜 Kernel-Classes ▷ Class ▷ subclass creation 프로토콜을 살펴보십시오.

비록 스퀵은 다중 상속을 지원하지 않지만, 버전 3.9 이후로, 연관성없는 클래스들의 기능을 공유하기 위한 traits라 지칭되는 메커니즘이 추가되었습니다. Traits는 상속과 관련되지 않은 다중 클래스들에서 재사용할 수 있는 메서드들의 Collection입니다. Traits를 사용하면 코드를 복사할 필요없이, 다양한 클래스들 사이의 코드를 하나의 코드로 공유할 수 있습니다.[3]

추상 메서드와 추상 클래스

추상클래스는 인스턴스화된다기보다는 하위분류로 존재하는 클래스입니다. 추상 클래스는 그것이 사용하는 모든 메서드를 정의하지 않는다는 면에서, 일반적으로 봤을때 완전한 클래스는 아닙니다. 이 경우 mission 메서드-다른메서드를 대상으로 하지만 메서드 자체(동작)가 정의되지 않은경우-는 추상메서드 라고 말합니다.

스몰토크는 메서드 또는 클래스를 abstract(추상)으로 지정하기 위한 문법은 별도로 없습니다. 관례적으로, 추상 메서드는 self subclassResponsibility의 표현식으로 만들어져 있습니다. 이것은 "마커 메서드(marker method)"로 알려져 있으며, 서브클래스가 메서드의 구체적인 내용을 정의해야한다는 것을 나타냅니다. Self subclassResponsibility 메서드는 항상 재지정되므로, 절대로 실행하면 안됩니다. 만약 여러분이 재지정 하는 것을 잊어버리고, 그 메서드를 실행하면, 예외가 적용될 것입니다.

[3] Ruby도 trait을 지원한다고 합니다. 이 내용을 보면 꽤나 오래전부터 지원되었던 느낌인데요.. Ruby에서의 지원예를 보고싶다면 여기에서 trait을 검색해 참고하면 되겠습니다.

클래스는, 자신의 메서드들 중의 하나가 추상적인 경우, 추상 클래스로 분류됩니다. 여러분이 추상 클래스의 인스턴스를 만드는 것을 제한하는것은 아무것도 없으며, 추상 메서드를 실행하기 전 까지는 모든 것이 동작합니다.

예시 : `Magnitude` 클래스

Magnitude 는 상호간 비교 가능 객체를 정의하는데 도움을 주는 추상 클래스입니다. Magnitude의 서브클래스는 반드시 메서드 `<`, `=`와 `hash`를 구현해야 합니다. 각각의 메시지를 사용하여, `Magnitude`에 `>`, `>=`, `<=`, `max:`, `min:`, `between:and:`등 객체를 비교하기 위한 다른 메서드를 정의합니다. 이들 메서드는 서브클래스에서 상속받습니다. 메서드 `<` 은 추상 메서드이며 메서드 5.7 에서 보시는 것처럼 정의합니다.

Method 5.7: `Magnitude>><`

```
Magnitude>>< aMagnitude
    "Answer whether the receiver is less than the argument."
    ↑self subclassResponsibility
```

반대로, 메서드 `>=` 는 메서드의 정의가 구체적입니다. `<`에 대해 정의하였습니다:

Method 5.8: `Magnitude>=`

```
>= aMagnitude
    "수신자가 인자보다 크거나 같은지 응답합니다."
    ↑(self < aMagnitude) not
```

다른 비교 메서드의 true 반환도 같습니다.

Character 는 `Magnitude`의 서브클래스입니다. Character클래스는 `<` 에 대한 자신의 버전을 갖기 위해 `subclassResponsibility` 메서드를 재정의합니다 (메서드 5.9 를 보세요). Character는 또한 메서드 `=`와 `hash`를 정의하며, 메서드 `>=`, `<=`, `~=` 과 기타 등등을 `Magnitude`로부터 상속 받습니다.

Method 5.9: `Character>><`

```
Character>>< aCharacter
    "Answer true if the receiver's value < aCharacter's value."
    ↑self asciiValue < aCharacter asciiValue
```

Traits

Trait은 상속이 필요없는 클래스의 동작을 포함할 수 있는 메서드의 Collection입니다. Trait은 여러 클래스가 유일한 super클래스를 쉽게 가지도록 하면서, 직접적 관련이 없는 클래스의 유용한 메서드를 공유하게 합니다.

새로운 trait을 정의하기 위해서는, 메시지에 의한 서브클래스 생성 템플릿을 Trait 클래스로 바꿔주기만 하면 됩니다.

Class 5.10: 새로운 *trait*을 정의하기

```
Trait named: #TAuthor
    uses: { }
    category: 'SBE--Quinto'
```

카테고리 SBE-Quinto 에서 TAuthor를 trait으로 정의하겠습니다. 이 trait은 다른 trait을 전혀 사용하지 않습니다. 일반적으로, `uses:`라는 키워드 인수의 다른 특성을 사용할 수 있게 하는 Trait 합성구문 을 지정할 수 있습니다. 여기서는 단지, 빈 array를 제공합니다.

author 메서드를 클래스 계층도에서 독립적으로 취급하며 다양한 class에서 사용하게 할거라면, 다음과같이 작성하면 됩니다:

Method 5.11: *Author* 메서드

```
TAuthor>>author
    "Returns author initials"
    ↑ 'on'    "oscar nierstrasz"
```

2장 장에서 정의한 `SBEGame` 클래스와 같이, 자체적으로 super클래스를 이미 보유한 클래스에서 이 trait을 사용할 수 있습니다. TAuthor를 사용하라고 지시하는 uses: 키워드 인자를 포함하기 위해 SBEGame의 클래스 생성 템플릿을 간단히 수정하겠습니다.

Class 5.12: *trait* 사용하기

```
BorderedMorph subclass: #SBEGame
    uses: TAuthor
    instanceVariableNames: 'cells'
    classVariableNames: ''
```

```
    poolDictionaries: ''
    category: 'SBE--Quinto'
```

이제 SBEGame 을 인스턴스화 하는경우, 예상대로 author 메시지에 반응합니다.

```
SBEGame new author  ⟶  'on'
```

Trait 합성 표현식은 + 연산자를 사용하여 여러 trait을 결합할 수 있습니다. 충돌이 생기는 경우(예를 들어, 여러 trait에서 같은 이름의 메서드를 정의할 경우), 이들 충돌 요소를 메서드 제거(- 기호)를 통해 깔끔하게 해결할 수 있으며, 또는 클래스 또는 trait의 메서드를 재 정의하여 해결할 수도 있습니다. 또한 (@ 기호) 메서드의 새 이름을 쓰려는 목적으로 메서드의 별칭*alias* 을 사용할 수도 있습니다

Traits는 시스템 커널에서도 사용합니다. Behavior 클래스는 좋은 예제입니다.

Class 5.13: *traits*을 사용하여 정의된 동작

```
Object subclass: #Behavior
    uses: TPureBehavior @ {#basicAddTraitSelector:withMethod: ⟶
                           #addTraitSelector:withMethod:}
    instanceVariableNames: 'superclass methodDict format'
    classVariableNames: 'ObsoleteSubclasses'
    poolDictionaries: ''
    category: 'Kernel--Classes'
```

위의 예에서 `addTraitSelector:withMethod:` 라고 이름을 지정한 `TPureBehavior`의 특성*trait* 에 정의된 `basicAddTraitSelector:withMethod:` 를 확인할 수 있습니다. Trait에 대한 지원은 현재 시스템 브라우저에 추가되고 있습니다.

5.5 모든 동작은 메시지를 보낼 때 일어난다

이 규칙은 스몰토크프로그래밍의 핵심이라 할 수 있습니다.

절차적 프로그래밍에서는, 호출자가 프로시저를 호출할 때 코드 일부를 선택합니다. 그리고 호출자는 이름을 이용해서 실행할 프로시저 또는 함수를 선택하죠.

객체지향 프로그래밍에서, 우리는 "메서드 호출"을 하지 않고, "메시지 전송"을 사용합니다. 이러한 용어의 선택은 중요합니다. 객체 지향 프로그래밍에서 각각의 객체는 그 자신이 각각 책임을 가집니다. 프로시저를 사용해서 객체에서 할일을 지시하지 않죠. 대신, 메시지를 보내어 어떤 일을 해달라고 정중하게 요청합니다. 메시지는 코드의 일부가 아닙니다: 단지 이름이고 인수의 목록일 뿐 입니다. 수신자는 요청받은 메시지의 내용에 따라 자신 스스로에게서 메서드를 선택하고 어떻게 응답할지를 결정합니다[4]. 각각의 객체가 동일한 메시지에 응답하기 위해 각자 메서드를 가지고 있기 때문에, 메세지를 받는 시점에서 메서드는 동적으로 선택됩니다.

```
3 + 4       ⟶  7      "send message + with argument 4 to integer
     3"
(1@2) + 4   ⟶  5@6    "send message + with argument 4 to point
    (1@2)"
```

결국, 메시지는 다양한 객체로 보낼 수 있으며 각각의 객체는 메세지에 응답하는 각각의 메서드를 가지게 됩니다. 우리는 $+\,4$ 메시지에 응답하는 방법인, Smallinteger 3 또는 Point 1@2 를 별도로 취급하지 않습니다. 각 클래스들은, +를 사용할 자체 메서드를 갖고 있으며, 그 메서드를 사용하여 $+\,4$ 라는 메시지에 응답합니다.

메시지 보내기를 기초로한 스몰토크모델의 결과로서 스몰토크는, 너무나 많은 일을 해야하는 절차적인 메서드보다는, 다른 객체에 작업을 넘기거나 작은 크기의 여러 메서드를 가지는 객체를 권장합니다.

Joseph Pelrine는 이 원리를 다음과 같이 간결하게 표현하였습니다:

[4] ta.onionmixer.net/wordpress/?p=172이부분의 내용을 참고

> 다른 녀석에게 맡길 수 있는 일을 일부러 하지 마세요[a]
>
> [a] 원문을 직역하면 "객체가 처리할 수 있는 요소를 다른 곳에 두지 마십시오" 이런 내용이 됩니다. 워낙 말이 괴악해서 의역을 진행했네요

수많은 객체 지향 언어에서는 객체에 대한 정적 및 동적 연산을 제공하지만, 스몰토크에서는 동적 메시지 전송만 존재합니다. 정적 클래스 연산이 없는 대신, 인스턴스, 클래스, 객체는 클래스에 메세지를 보내는 방법을 사용할 수 있습니다.

스몰토크에서 거의 모든 작업은 메시지 전송으로 이루어집니다. 하지만 몇가지 일부동작은 반드시 필요합니다:

- 변수 선언은 메시지 전송이 아닙니다. 사실, 변수 선언은 실행되는 요소도 아닙니다. 변수를 선언하면 단지 객체 참조에 사용할 공간을 할당하는 일만 일어날 뿐입니다.
- 할당은 메시지 전송이 아닙니다. 변수 할당은 변수를 선언한 범위내에서 "변수가 존재하도록" 하는 작업입니다.
- 반환은 메시지 전송이 아닙니다. 계산된 결과를 간단하게 송신자로 반환하는 작업입니다.
- 프리미티브는 메시지 전송이 아닙니다. 프리미티브는 가상머신에서 실행됩니다.

위의 몇 가지 예외를 제외하고, 거의 모든 작업은 메시지 전송으로 이루어집니다. 특별히, 스몰토크의 클래스에서는 에서는 "공용 필드(public영역)"가 존재하지 않기 때문에, 다른 객체의 인스턴스 변수를 업데이트 하는 방법은, 객체에 메시지를 보내어 객체 내부의 필드를 업데이트하도록 메시지를 보내는것 외에는 없습니다. 물론, 객체의 모든 인스턴스 변수를 위해 setter와 getter 메서드를 제공하는 것은 좋은 객체지향 방식이 아닙니다. Joseph Perlrine은 이 내용을 매우 훌륭하게 표현했습니다.

누구도 당신의 데이터를 함부로 손대지 못하게 하십시오.

5.6 메서드 탐색은 상속 관계를 따릅니다

객체가 메시지를 받을 때는 어떤 일이 일어날까요?

과정은 꽤 단순합니다: 수신자의 클래스는 메시지를 처리하기 위해 클래스가 가진 메서드중 사용할것을 검색합니다. 만약 수신자 클래스에 메세지에 적합한 메서드가 없다면, 클래스는 상위클래스에 요청하고 찾을때까지 상속관계를 따라 클래스를 거슬러 올라갑니다. 메서드를 발견 했을 때, 메서드의 매개변수로 인자를 묶어 한덩이로 만들고 가상머신은 묶인것을 실행합니다.

설명과는 달리, 과정이 그리 복잡하지는 않습니다. 하지만, 몇몇 질문들은 중요한점을 알려주고 있죠.

- 메서드가 명확하게 값을 반환하지 못할때는 어떤 일이 일어나나요?
- 클래스가 super클래스의 메서드를 다시 구현하면 어떤 일이 일어나나요?
- self와 super의 메시지 전송상 차이점은 무엇입니까?
- 메서드를 찾지 못하면 어떤 일이 일어나나요?

여기서 보여드린 메서드 검색규칙은 개념일 뿐입니다: 가상 머신 구현체[5]는 메서드 검색 방식의 속도를 늘리기 위해 모든 종류의 기술과 최적화를 사용합니다. 가상 머신 구현체는 이런 실질적인 최적화 작업들을 하지만, 사용자는 여기서 제시한 규칙과 다른 동작을 전혀 확인할 수 없습니다.

먼저 기본적인 검색 전략을 살펴보고, 그 다음 추가적인 질문에 대해 생각해 보겠습니다.

[5] 이 경우는 CogVM을 말하는게 되겠죠?

메서드 검색

우리가 EllipseMorph의 인스턴스를 만든다고 가정해 봅시다.

```
anEllipse := EllipseMorph new.
```

만약 defaultColor 라는 객체 메시지를 보낸다면, Color yellow: 를 결과로 얻을 수 있습니다.

```
anEllipse defaultColor  ⟶  Color yellow
```

클래스 EllipseMorph는 defaultColor를 실행하므로, 적합한 메서드는 수신자 클래스 내부에서 즉시 발견됩니다.

Method 5.14: 로컬에서 실행된 메서드

```
EllipseMorph>>defaultColor
    `answer the default color/fill style for the receiver'
    ↑ Color yellow
```

반대로, 만약 메시지 openInWorld 를 Ellipse에 보낸다면, 이 메서드는 즉시 발견되지 않습니다, 그 이유는 클래스 EllipseMorph는 openInWorld를 구현하지 않았기 때문이죠. 그러므로 메서드 검색은, openInWorld 메서드를 클래스 Morph에서 발견할 때까지, super클래스 BorderedMorph 그리고 다른 클래서에서도 계속됩니다. (그림 5.2 를 보십시오)

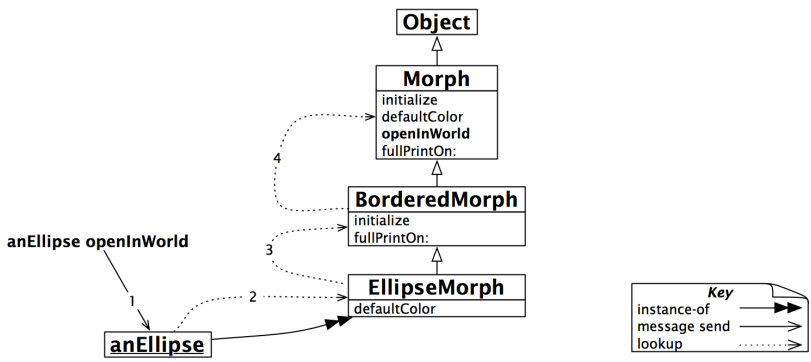


그림 5.2: 메서드 lookup은 상속받은 상속관계를 따릅니다

self 반환하기

`EllipseMorph>>defaultColor` (메서드 5.14)는 명쾌하게 `Color yellow`를 반환하지만, `Morph>>openInWorld` (메서드 5.15)는 아무것도 반환 하지 않음을 확인할 수 있습니다.

Method 5.15: 상속된 메서드

```
Morph>>openInWorld
    `Add this morph to the world. If in MVC, then provide a Morphic
    window for it.'
    self couldOpenInMorphic
        ifTrue: [self openInWorld: self currentWorld]
        ifFalse: [self openInMVC]
```

실제로, 메서드는 항상 값(value)으로 메시지에 응답합니다 - 여기서 반환되는 값(value)은 당연히 객체입니다. 응답은 메서드에서 ^ construct의 형식으로 이미 정의되었겠지만, 실행된 시점에서 ^를 실행하지 않고, 메서드의 끝부분에 도달했다면, 메서드는 여전히 값(value)을 내놓게 됩니다: 이 경우 메서드는 메시지를 수신한 객체(수신자) 자체를 반환합니다. 이런것들을 가리켜 "self 응답" 메소드 라고 합니다. 스몰토크에서, 의사 변수 self 는 오히려 Java 에서 `this`과 같기 때문에 그렇습니다.

여기서 openInWorld(메서드 5.15)는 openInWorldReturnSelf(메서드 5.16)와 동일하게 취급함을 보여주고 있습니다.

Method 5.16: 자신을 분명하게 반환하기

```
Morph>>openInWorld
    "Add this morph to the world.  If in MVC,
    then provide a Morphic window for it."
    self couldOpenInMorphic
        ifTrue: [self openInWorld: self currentWorld]
        ifFalse: [self openInMVC].
    ↑ self      "Don't do this unless you mean it!"
```

여러분이 뭔가를 분명하게 반환하려면, 발신자가 관심있어 할 만한 것을 반환하며 의사소통을 하겠죠. 당신이 정확하게 self를 반환한다는 의미는, 발

신자는 반환값을 사용할게 될 거라고 예측하고 진행을 하는 것과 같습니다.[6] 이런 프로그래밍 방법은 self를 정확하게 반환하는 좋은 방법도 아닐뿐더러 여기서 다룰만한 사례도 아닙니다.

이 `^self`는 스몰토크에서 일반적인 용어이며, Kent Beck이 "흥미로운 반환값" [4]이라고 언급한 것입니다.

> 전송자가 값을 사용하기를 원할 때에만 값을 반환하면 됩니다.

재지정 및 확장 (Overriding and extension)

그림 5.2 에 있는 EllipseMorph 클래스 계층도를 다시 살펴보면, Morph 클래스와 EllipseMorph 클래스는 둘 다 defaultColor 를 구했음을 알 수 있습니다. Morph new openInWorld 로 새 Morph를 만들면 타원은 원래 노랑색이어야 하지만 파랑색으로 만들어지는걸 확인할 수 있죠.

이런경우를 defaultColor 메서드는 Morph로부터 상속받은 다음 EllipseMorph가 재지정*override* 했다고 말합니다. defaultColor의 경우처럼 상속된 메서드는, anEllipse의 관점에서는 더이상 존재하지 않는것이 됩니다.

가끔 서브클래스를 만들 때 상속받은 메서드를 재정의하는것이 아니라 원래 있던 메서드의 기능에 새로운 기능을 덧붙이고 싶을때가 있습니다[7]. 재지정된 메소드의 원래기능에 새로운 기능을 추가해서 사용하고 싶은 경우가 있기 때문이죠. 스몰토크에서, 단일 상속을 지원하는 많은 객체 지향 언어들처럼, 이 작업 역시 super send를 사용하면 가능합니다.

initialize 메서드는 이런 메커니즘에 있어서 대단히 중요한 예가 됩니다. 클래스의 새 인스턴스를 초기화 하는 때가 언제든지간에, 상속 인스턴스 변수를 초기화 하는 것 역시 대단히 중요합니다. 그러나, 이를 처리하는 방법에 대한

[6] 수신자와 발신자가 서로에 대해 명확하게 알아야 동작되는 상황을 피하라는 의미겠죠.

[7] 자식 클래스를 구현함에 있어 새로운 기능을 추가하면서 상속-재지정된 method 를 호출할 수 있다는 말입니다.

내용은 이미 상속 관계상 각각의 super클래스에 있는 initialize 메서드에 이미 작업되어 있습니다. 서브클래스는 상속 인스턴스 변수를 초기화 할 권리가 없습니다!

그러므로 서브클래스 스스로의 initialize 를 진행하기 전에 super initialize 를 먼저 진행하는 것은 좋은 방법입니다:

Method 5.17: *Super initialize*

```
BorderedMorph>>initialize
    "initialize the state of the receiver"
    super initialize.
    self borderInitialize
```

> initialize 메서드는 항상 super initialize(상위 초기화)를 보내며 시작합니다.

self send와 super send

상속 받은 메서드의 동작을 재지정려면 super send가 필요합니다. 그러나 상속 여부와 관계 없이 일반적으로 메서드를 구현하려면 self send 를 사용하면 됩니다.

self send는 super send와 어떤 차이가 있을까요? self와 super는 둘 다 메시지의 수신자(자기 자신)를 의미합니다. 다른점이 있다면 메서드를 검색할때 뿐입니다. self가 메시지를 받았을때 메서드의 검색을 수신자의 클래스에서 시작한다면, super는 super send가 발생된, 현재클래스의 상위클래스부터 검색을 시작합니다.[8]

super 자체는 super클래스를 나타내는것이 아니라는 것을 주의하세요! 사실 이렇게 오해할만 하기도 합니다. 또한 super 를 사용하면 수신자의 super 클래스 내부에서 검색을 한다고 잘못 생각할 수도 있습니다.

임의의 모프로 보내질 수 있는 initString 메시지가 있다고 가정해보겠습니다:

[8] self와 super에 대한 내용은 이 부분 을 찾아보면 좀 더 자세히 알 수 있습니다.

```
anEllipse initString  ⟶  '(EllipseMorph newBounds: (0@0 corner:
    50@40) color: Color yellow) setBorderWidth: 1 borderColor:
    Color black'
```

위의 경우 반환되는 값은 모프의 재생성 작업을 할 수 있는 문자열입니다. self와 super send의 조합을 이용한다면 어떻게 해야 정확한 결과를 얻을 수 있을까요? 먼저, 그림 5.3 에 보이는 것 처럼, anEllipse initString은 메서드 initString이 클래스 Morph에서 발견되도록 합니다.

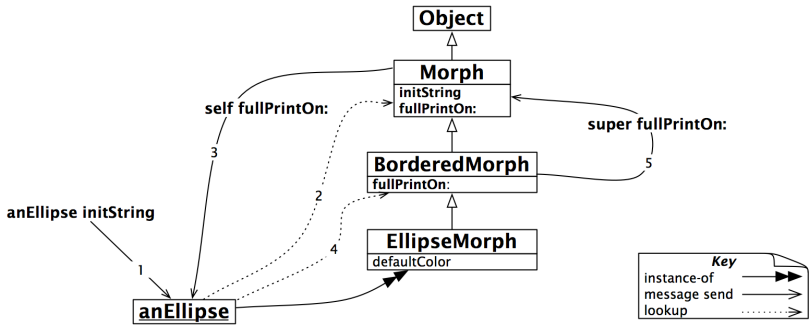


그림 5.3: self 와 super 전송

Method 5.18: *A* self *send*

```
Morph>>initString
    ↑ String streamContents: [:s | self fullPrintOn: s]
```

메서드 `Morph>>initString` 은 `fullPrintOn:`의 `self send` 를 수행합니다. 이 작업은 다시 원래의 수신자의 클래스인 EllipseMorph 클래스에서부터 두 번째 검색을 시작하게 하며 BorderedMorph 클래스에서 `fullPrintOn:` 를 찾게됩니다. (그림 5.3 을 다시 보십시오)

여기서 대단히 중요한 점은, self send가 사용되면 수신자 anEllipse의 클래스인 EllipseMorph 클래스에서 메서드 찾기를 다시 시작하도록 만든다는 것입니다.

> self send는 수신자의 클래스에서 동적 메서드 검색을 시작하게 합니다.

Method 5.19: super *send*와 self *send*의 조합

```
BorderedMorph>>fullPrintOn: aStream
    aStream nextPutAll: '('.
    !\textbf{super fullPrintOn: aStream.}!
    aStream nextPutAll: ') setBorderWidth: '; print: borderWidth;
        nextPutAll: ' borderColor: ' , (self colorString:
    borderColor)
```

super send가 사용되었기 때문에, Morph 클래스에서 super send가 실행된 클래스의 super 클래스에서 메서드 검색을 시작합니다. 그 다음, `Morph>>fullPrintOn:`를 찾고 처리합니다.

super 에 의한 메서드 검색이 수신자의 super 클래스에서 시작되지 않음에 주의합니다. 만약 super클래스에서 시작한다면 무한반복으로 수렴하는, BorderedMorph에서의 검색이 시작됩니다[9].

> super send는 super send를 실행하는 메서드를 가진 클래스의 super 클래스에서 시작되는 정적 메서드 검색을 진행합니다.

만약 당신이 super send와 그림 5.3 에 있는 initString의 탐색과정을 주의깊게 보셨다면, `super`의 바인딩은 정적이라는걸 아셨을 겁니다: 모든문제는 super send가 들어있는 소스코드를 가진 클래스에 있습니다. 이런 super와는 반대로 self의 바인딩은 동적입니다: self는 항상 현재 실생중인 메세지의 수신자를 의미하며 self로 전송된 모든 메세지들에 대한 메서드 검색은 수신자 클래스 자신부터 시작된다는 의미가 됩니다.

[9] super send가 호출된 메서드를 소유한 클래스의 상위클래스에서 메서드를 검색하기때문에 메서드 검색에 있어 무한루프에 빠지지 않는다는 의미

이해할 수 없는 메시지

원하는 메시지를 발견하지 못하면 어떤 일이 일어날까요?

메시지 foo를 anEllipse클래스에 보낸다고 가정하겠습니다. 첫번째로 일반적인 메서드 찾기 작업은 상속 관계를 따라 Object 클래스까지(또는 ProtoObject 클래스까지도) 이 메서드검색을 위해 이동됩니다. 메서드를 발견하면, 가상 머신은 객체에 `self doesNotUnderstand: #foo`를 보냅니다. (그림 5.4 를 봐주세요)

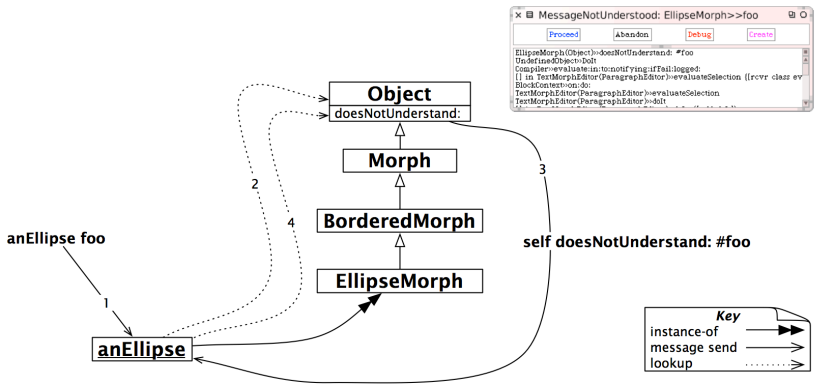


그림 5.4: 메시지 foo를 이해할 수 없습니다

이런 메서드 검색에 대한 반응동작은 완벽하게 일반적인, 동적 메시지 전송 이기 때문에, 메서드 검색은 클래스 EllipseMorph 클래스부터 다시 시작하지만, 이번에는 `doesNotUnderstand:` 라는 메서드를 검색합니다. 그리고 보이는 것처럼, 객체는 `doesNotUnderstand:` 메서드를 실행합니다:이 메서드는 새로운 MessageNotUnderstood 객체를 만들고, 이렇게 만들어진 `MessageNotUnderstood` 객체는 현재의 진행 상황에서 디버거의 시작을 가능하게 합니다.

왜 이렇게 누가봐도 오류인 상황을 취급하기 위해 이 복잡한 처리과정을 거쳐야 할까요? 글쎄요, 이런 과정은 개발자에게 이런식으로 발생하는 오류를 가로채서 다른 작업을 할 수 있도록 해주는 쉬운 방법을 제공합니다. 대안을 원하는 경우, 모든 객체의 서브 클래스에서 메서드 "doesNotUnderstand" 를 쉽게 재지정하고, 오류를 취급할 수 있는 다른 방법을 제공할수도 있습니다.

사실, 이런 방법은 어떤 객체에서 다른 객체로 메시지 전달을 자동 위임 할 수 있도록 구현하는 쉬운 방법중 하나가 될 수 있습니다. Delegator 객체는 수신자 자신이 이해하지 못하는 모든 메시지를 처리할 책임이 있는 다른 객체에 넘기거나 자체적으로 오류를 발생시킵니다!

5.7 공유 변수

이제 다섯 가지 규칙이 잘 적용되지 않는 스몰토크의 다른부분을 살펴보겠습니다 : 공유변수*Shared variables* 입니다.

스몰토크에는 3가지 종류의 공유변수가 있습니다 : (1) 전역 공유 변수 (2) 인스턴스와 클래스 사이에서 공유되는 변수 (클래스 변수) 그리고 (3) 클래스의 그룹 (Category) 사이에서 공유된 변수 (pool 변수) 들. 이 세가지의 공유변수들은 사용자에게, 이 변수들은 공유된 변수라는 것을 알려주기 위해 대문자로 시작됩니다.

전역 변수

스퀵에서, 모든 전역 변수는 클래스 SystemDictionary의 인스턴스로서 실행되며, Smalltalk라고 불리는 네임스페이스에 저장됩니다. 스몰토크의 전역 변수는 어디서나 접근이 가능합니다. 모든 클래스는 전역 변수 같은 형식의 이름을 가지게 됩니다.[10] 게다가, 일부 이름은 특별하거나 일반적으로 유용한 객체를 이름지을 때 사용합니다.

Transcript 변수는 스크롤 창에 데이터를 기록하는 스트림인 TranscriptStream 인스턴스의 변수명 입니다. 다음 코드는 Transcript에 일부 정보를 표시하고 표시 커서를 다음 줄로 이동시킵니다.

```
Transcript show: 'Squeak is fun and powerful' ; cr
```

do it 을 클릭하기 전에, *Tool* 플랩 에서 Transcript 를 드래그하여 트랜스크립트를 열어야 합니다.

[10] 첫 문자가 대문자인 규칙을 말하는듯 합니다.

HINT *Transcript*에 데이터를 기록하는 작업은 *Transcript* 창이 열렸을 때 더 느려집니다. 따라서, *Transcript*에 데이터를 기록하는 동작이 느리다고 느껴진다면, *Transcript* 창을 닫는 것을 고려해 보십시오.

다른 유용한 전역 변수들

- Smalltalk는 스몰토크자신을 포함해서, 모든 전역 변수를 정의하는 SystemDictionary의 인스턴스 입니다. SystemDictionary 라는 dictionary의 핵심은 스몰토크코드에서 전역 오브젝트를 이름짓는 심볼입니다. 아래 예를 보겠습니다.

```
Smalltalk at: #Boolean  ⟶  Boolean
```

  스몰토크자체가 전역 변수이므로,

```
Smalltalk at: #Smalltalk ⟶ a SystemDictionary(lots of
globals)
```

  그리고

```
(Smalltalk at: #Smalltalk) == Smalltalk  ⟶  true
```

- Sensor는 EventSensor의 인스턴스이며, Squeak으로 들어가는 입력들을 처리합니다. 예를 들어, Sensor keyborad는 키보드에 다음 문자 입력에 응답하며, Sensor mousePoint는 Point는 현재 마우스 위치를 알려주는 반면에, 만약 왼쪽 쉬프트 키가 눌려 있는 상태인 경우 Sensor leftShiftDown은 true로 응답됩니다.
- World는 화면을 표시하는 PasteUpMorph 클래스의 인스턴스입니다. World bounds는 전체 화면 공간을 정의하는 직사각형의 정보를 반환하며, 화면의 모든 모프는 World의 하위모프가 됩니다.
- ActiveHand는 HandMorph의 현재 인스턴스이며, 커서의 그래픽 표현입니다. ActiveHand의 서브모프는 마우스로 드래그한 모든 것을 잡습니다.

- Undeclared는 또다른 dictionary입니다 - Undeclared dictionary에는 모든 미리 선언되지 않은 변수들이 포함됩니다. 만약 당신이 선언하지 않은 함수를 참조하는 메서드를 작성하게 된다면, 시스템 브라우저는 대개의 경우 선언되지 않은 함수를 선언하도록 알려줍니다. 하지만, 만약 나중에 해당되는 선언문을 삭제한다면, 코드는 선언문이 지워졌기 때문에, dictionary에는 있지만 실제로는 선언되지 않은 변수들을 참조하는 상황이 됩니다. Undeclared dictionary를 살펴보면 가끔 이상한 동작을 하는경우에 대해 도움을 얻을 수 있는 경우도 있습니다.

- SystemOrganization은 SystemOrganizer 클래스의 인스턴스입니다: SystemOrganization은 패키지별로 클래스들의 그룹을 기록합니다. 좀 더 정확하게 말하자면, 이것은 클래스들의 이름을 그룹으로 만들어 분류(categorizes)합니다. 그 덕분에 다음과 같은 코드로 결과를 얻을 수 있습니다[11].

```
SystemOrganization categoryOfElement: #Magnitude  ⟶
#'Kernel--Numbers'
```

현재 진행하실 연습에서는 전역 변수의 사용을 엄격히 제한해주세요. 인스턴스 변수 또는 클래스 변수를 사용하거나, 전역 변수들에 접근하기 위해서 클래스 메서드를 제공하는 방법이 더 좋은 방법입니다. 정말로, 스퀵을 현재 상태로 바닥부터 구현한다면, 클래스가 아닌 대부분의 전역번수들은 아마도 Singleton 으로 대체되었을 겁니다.

보통 전역 요소를 정의할때는 머릿 글자를 대문자로 하였지만 아직 선언하지 않은 식별자*undeclared identifier* 를 마우스로 영역선택하고 `do it`을 실행하면 됩니다. 파서*parser* 는 그 다음, 사용자를 위해 선언할 전역 변수를 제공할 것입니다. 만약 프로그램문장 으로 전역 변수를 정의하기 원한다면, `Smalltalk at: #AGlobalName put: nil` 을 실행하면 됩니다. 이렇게 만들어진 변수를 제거하려면, `Smalltalk removeKey: #AGlobalName` 를 실행합니다.

[11] 이 부분에 대한 내용은 여기를 보면 아주 조금 더 자세하게 볼 수 있습니다

클래스 변수

때로는 클래스 인스턴스와 클래스 자신 양쪽에 무슨 데이터든 공유할 필요가 있습니다. 이런 작업에는 클래스 변수를 사용하면 됩니다. 클래스 변수라는 용어는 변수의 수명이 클래스의 수명과 같음을 나타냅니다. 하지만 이런 의미로는 좀 모자라는것이, 클래스 변수는 그림 5.5 에 보이는 것처럼 클래스 자체 뿐만 아니라 클래스의 모든 인스턴스들 사이에서도 그 내용이 공유되기 때문입니다. 실제로, 클래스 변수 보다 좀 더 나은 이름을 쓴다면 '공유 변수'가 되겠죠. 왜냐하면, 이 이름이 좀 더 명확하게 클래스변수의 역할을 설명해 주기 때문이며 또한, 변수들이 수정되는 경우, 사용상 위험성을 사용자에게 경고하기 때문입니다.

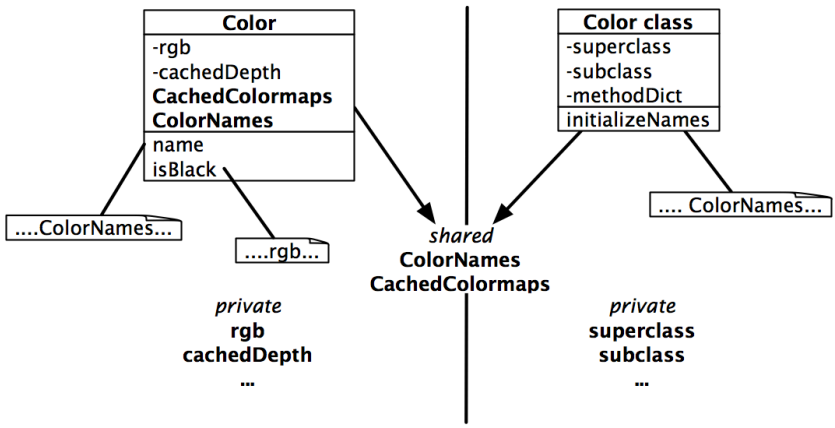


그림 5.5: 다양한 변수들에 접근하는 인스턴스와 클래스 메서드

그림 5.5 에서, 우리는 rgb와 cachedDepth가 Color 인스턴스의 인스턴스 변수인 것을 알 수 있으며, Color 인스턴스만 접근이 가능한 존재라는걸 알 수 있습니다[12]. 또한 superclass, subclass, methodDic 과 다른 것들이 color 클래스의 클래스 인스턴스 변수임을 알 수 있습니다. 말하자면, 이런 클래스 인스턴스 변수들은 오직 color 클래스에서만 접근이 가능다는 내용입니다.

[12]스몰토크에서 변수는 일반적으로 Private으로 취급되기 때문입니다

그러나 약간 새로운 점도 있습니다: ColorNames와 CachedColormaps는 Color 를 위해 정의된 class 변수입니다. 이 변수들의 이름중 맨 앞글자가 대문자인걸 보면 이것들이 공유변수라는걸 암시합니다. 사실, Color의 모든 인스턴스는 이 공유변수들에 접근가능할 뿐만 아니라, Color 클래스 자체와 그 클래스의 모든 서브클래스도 공유변수에 대한 접근이 가능합니다. 인스턴스 메서드와 클래스 메서드 양쪽다, 이 공유변수에 접근이 가능합니다.

클래스 변수는 클래스 정의 템플릿에 선언됩니다. 예를 들어, Color 클래스에서는 색상을 빨리 만들기 위한 수많은 클래스 변수를 정의하고 있으며, 내용은 아래에서 볼 수 있습니다. (class 5.20)

Class 5.20: *Color*와 *Color*의 클래스 변수

```
Object subclass: #Color
  instanceVariableNames: 'rgb cachedDepth cachedBitPattern'
  classVariableNames: 'Black Blue BlueShift Brown CachedColormaps
    ColorChart
  ColorNames ComponentMask ComponentMax Cyan DarkGray Gray
  GrayToIndexMap Green GreenShift HalfComponentMask
    HighLightBitmaps
  IndexedColors LightBlue LightBrown LightCyan LightGray
    LightGreen
  LightMagenta LightOrange LightRed LightYellow Magenta
    MaskingMap Orange
  PaleBlue PaleBuff PaleGreen PaleMagenta PaleOrange PalePeach
    PaleRed
  PaleTan PaleYellow PureBlue PureCyan PureGreen PureMagenta
    PureRed
  PureYellow RandomStream Red RedShift TranslucentPatterns
    Transparent
  VeryDarkGray VeryLightGray VeryPaleRed VeryVeryDarkGray
  VeryVeryLightGray White Yellow'
  poolDictionaries: ''
  category: 'Graphics----Primitives'
```

클래스 변수인 ColorNames는 자주쓰이는 색상의 이름을 포함하고 있는 배열입니다. 이 배열은 color의 모든 인스턴스에서 공유되며, 배열의 서브클래스는 TranslucentColor 입니다. 이 배열은 모든 인스턴스와 클래스 메서드들로 부터 접근 가능합니다.

클래스 변수인 ColorNames는 일단 `Color class>>initializeNames` 에서

초기화되지만, Color 클래스의 인스턴스들이 접근(사용)합니다. 메서드 `Color>>name`은 색상의 이름을 찾기 위해 ColorNames 변수를 사용합니다. 모든 색상의 이름이 있는건 아니라서, 인스턴스 변수에 대부분 색상의 이름을 추가하는건 좋지 않습니다.

클래스 초기화

클래스 변수를 보면 이런 의문이 생깁니다: 어떻게 초기화하지? 이런 질문에 대한 해결책중 하나는 게으른초기화(lazy initialization) 입니다. 아직 초기화가 되지 않은 변수에 접근할 때 초기화를 진행하는 접근자 메서드를 도입하면 가능합니다. 하지만 게으른 초기화는 언제나 접근자를 사용해야 하며 클래스 변수를 직접 사용하면 안된다는 의미가 됩니다. 더욱이 접근자 전송과 초기화 테스트의 부담을 보태게 되죠. 또한 이 방법은 클래스 변수를 사용하는 이유을 없애버립니다, 왜냐하면 사실상 클래스변수가 더 이상 공유 되지 않기 때문입니다.

Method 5.21: `Color class>>colorNames`

```
Color class>>colorNames
    ColorNames ifNil: \href{self}{initializeNames}.
    ↑ ColorNames
```

게으른초기화 외의 다른방법은, 클래스 메서드 초기화를 재지정*override* 하는 것입니다.

Method 5.22: `Color class>>initialize`

```
Color class>>initialize
    !\ldots!
    self initializeNames
```

만약 초기화를 재지정하는 방법을 쓴다면 initialize 메서드를 정의한다음 실행해야 할 필요가 있다는걸 알고있어야 하는데 Color initialize 를 실행하는것처럼 하면 됩니다. 비록 클래스의 코드가 메모리에 로드될때 (시스템브라우저의)클래스측면에서 볼 수 있는 초기화 메서드들이 자동으로 실행되지만, 시스템 브라우저 에서 처음 입력하고 편집 및 컴파일했을때도 자동으로 실행되는건 아닙니다.

Pool 변수

Pool 변수는 상속 관계가 없을 수 있는 수많은 클래스사이에 공유되는 변수입니다. Pool 변수는 원래 pool dictionary에 저장됩니다; 하지만 지금은 단독 클래스(SharedPool의 서브클래스)의 클래스 변수처럼 정의해야 합니다. 우리는 이 변수를 사용하지 않기를 권합니다. 왜냐하면 매우 특히한 상황에서만 Pool변수가 필요하기 때문이죠. 여기서 Pool변수를 설명하는 이유는 코드를 분석할때 도움이 되도록 하기 위함입니다.

pool 변수에 접근하는 클래스는 접근을 원하는 클래스 정의에서 반드시 pool을 언급해야 합니다. 예를 들어, Text 클래스는 자신이 pool dictionary인 TextContansts를 사용하고 있다는걸 선언하며, TextContansts 라는 Pool 변수는 CR과 LF와 같은 모든 문자 상수를 포함합니다. TextContansts dictionary는, 예를 들어, 캐리지 리턴*carriage return* 문자값을 가진 Character cr에 묶인 `key #CR`등을 가지고 있습니다.[13]

Class 5.23: `Text` 클래스에 있는 *Pool dictionary*

```
ArrayedCollection subclass: #Text
        instanceVariableNames: 'string runs'
        classVariableNames: ''
        !\textbf{poolDictionaries: 'TextConstants'}!
        category: 'Collections--Text'
```

이러한 poolDictionary의 선언은, class 5.23 에 있는 Text 클래스의 메서드가 메서드 내용의 dictionary 키로 직접*directly* 접근하는 것을 허용합니다. 예를 들어, 다음과같은 메서드를 작성할 수 있다는 얘기죠.

Method 5.24: `Text>>testCR`

```
Text>>testCR
  ↑ CR == Character cr
```

다시 말하지만, 우리는 당신이 pool 변수와 pool dictionary 사용을 피하실 것을 권장하는 바입니다.

[13] system browser에서 해당되는 class를 선택한후 instance버튼을 눌러 인스턴스 메서드를 확인하면 내용을 정확히 알 수 있습니다.

5.8 5장 요약

스퀵의 객체 모델은 모두 단순하며 일관성이 있습니다. 모든 요소는 객체이며, 대부분 동작은 메시지 전송으로 이루어집니다.

- 모든 요소는 객체이며, 정수와 같은 프리미티브 엔티티 객체이고, 클래스는 일등급 객체 입니다.
- 모든 객체는 클래스의 인스턴스이며, 클래스는 전용 인스턴스 변수와 그 변수의 인스턴스들의 동작을 통해, 스스로의 인스턴스 구조를 정의합니다. 각 클래스는 그 클래스의 메타클래스가 가진 고유한 인스턴스입니다. 클래스 변수은 클래스와 그 클래스의 모든 인스턴스들에 의해 공유되는 private 변수입니다. 클래스는 그 클래스의 인스턴스가 가진 인스턴스 변수에 직접 접근할 수 없으며, 인스턴스는 그 인스턴스의 클래스가 가진 인스턴스 변수에 접근할 수 없습니다. 접근자는 반드시 접근자가 필요할때만 정의되어야만 합니다.
- 모든 클래스는 super 클래스를 갖고 있습니다. 단일 상속 계층도의 뿌리*root* 는 `ProtoObject` 입니다. 사용자가 정의하는 클래스는 객체 또는 그 객체의 서브클래스에서 상속됩니다. 추상 클래스들을 정의하는 어떤 프로그램식도 존재하지 않습니다. 추상 클래스란, 실행이 표현식 self subclassResponsibility 로 구성된 추상 메서드를 가진 클래스 입니다. 비록 스퀵이 단일 상속만을 지원하지만, traits 로서 메서드 패키지를 만들어, 메서드들의 실행을 공유하는 작업은 어렵지 않습니다.
- 모든 동작은 메시지 전송으로 이루어집니다. 스몰토크에서는 "메서드 호출"을 하지 않고, "메시지 전송"을 사용합니다. 메시지를 전송하면, 수신자는 그 메시지에 응답하기 위해 수신자 자신의 메서드 중 알맞는 것을 선택합니다.
- 메서드 탐색은 상속 관계를 따라 수행합니다. super send는 정적이며, 메서드 탐색은 super send를 작성한 장소에 있는 클래스의 super

class에서 시작됩니다. 반면에, self send는 동적이며, 메서드 탐색은 수신자의 클래스에서 다시 시작합니다.

- 3가지 종류의 공유 변수들이 있습니다. 전역 변수는 스몰토크시스템 어디에서나 접근할 수 있습니다. 클래스 변수들은 클래스, 그 클래스의 서브클래스 그리고 그 클래스의 인스턴스 사이에서 공유됩니다. Pool 변수는 클래스 초기화 부분에서 poolDictionary를 설정한 클래스들 사이에서만 공유됩니다. 공유 변수는 최대한 사용하지 말아주세요.

제 6 장

스퀵프로그래밍 환경

이 장의 목표는 스퀵프로그래밍 환경에서 어떻게 프로그램을 개발하는지 보여드리는 것입니다. 여러분은 이미, 시스템 브라우저를 사용하여 메소드를 정의하는 방법을 보셨겠지만, 이 6장 은 시스템 브라우저에 대해 더 많은 기능을 알려주고, 다른 브라우저의 기능 또한 소개해 드릴 것입니다.

물론, 당신이 기대했던 것만큼 당신의 프로그램이 늘 잘 작동하는건 아니라는것도 알게될겁니다. 스퀵은 훌륭한 디버거를 갖고 있지만, 대부분의 강력한 도구들처럼, 처음 사용시에 헷갈릴수 있습니다. 6장 에서는 디버깅 세션과 디버거의 몇몇 기능들을 알려드리게 됩니다.

스몰토크만 가지고있는 고유한 기능 중의 하나는, 당신의 프로그래밍을 동작시킬때, 스퀵환경은 정적인 텍스트상태가 아니라 살아있는 객체들의의 환경이라는걸 기억해 주세요. 프로그램이 동작하는 환경이 살아있는 객체가 된다라는것은 당신이 프로그래밍을 하는 동안 매우 신속한 피드백을 얻을 수 있게 함으로서 매우 생산적인 환경에 있다는것을 의미합니다. 사용자가 살아있는 객체를 살펴보고 바꿀 수 있게 하는 두가지 도구가 있습니다 — Inpector와 탐색기*explorer* 입니다.

파일과 텍스트 에디터를 사용하는 것보다 살아있는 객체의 환경에서 프로그래밍을 한후 결과물을 스몰토크로부터 밖으로 내보내고싶다면, 어떤 작업이

던 선택해서 진행해야 합니다. 모든 스몰토크파생 언어에서도 지원하는 프로그램을 내보내는 작업에 대한 전통적인 방법은, file out 또는 change set을 이용해서 다른시스템에서 import할 수 있는 기본인코딩된 텍스트파일을 만드는 것입니다. 스퀵에서 사용하는 새로운 방법은 당신의 코드를 서버의 버전별저장소로 업로드하는겁니다. 몬티텔로라고 하는 도구로 이런작업을 진행할 수 있으며, 이렇게 저장소로 업로드하는 방법은 팀별로 일을할때 보다 강력하고 효율적입니다.

마지막으로, 작업을 하는 도중에 스퀵에서 버그를 발견할 수 있습니다. 버그를 보고하는 방법과 버그 수정을 제출하는 방법을 설명하도록 하겠습니다.

6.1 개요

스몰토크와 최신 그래픽 인터페이스는 같이 개발되었습니다. 1983년 스퀵이 처음으로 대중에게 공개되기 전에도, 스몰토크는 자체적으로 구현된(X-windows등이 아닌) 그래픽 개발 환경을 갖고 있었으며, 모든 스몰토크개발은 이런 자체 그래픽 개발 환경에서 이루어졌습니다. 이제 스퀵의 주요 도구들을 살펴봄으로써 이 장을 시작하겠습니다. 스퀵의 모든 도구들은 Squeak-ByExample.org 에서 *Tools* 플랩 의 외부로 끌고갈 수 있습니다. 여러분은 자신만의 개인 설정을 사용하여 Tools 플랩에 마우스를 포인터를 올려 놓거나 스퀵메인 창의 오른쪽 끝의 오렌지 색 탭을 클릭하여 도구를 열수 있습니다.

- 브라우저 (Browser) 는 가장 기본이 되는 개발 도구입니다. 여러분은 자신의 클래스와 메서드를 만들고 정의하고 준비하는 작업에 이 기본 개발 도구를 사용할 것입니다. 이 도구를 사용하여, 모든 라이브러리 클래스 구석구석을 검색할 수 있습니다: 스몰토크에서는 소스 코드가 프로그램과 별도의 파일들로 저장되는 다른 환경들과 달리, 모든 클래스와 메서드를 이미지에 저장합니다.

- 메시지 이름 (the Message Names) 도구는 특별한 선택자*selector* 또는 하위문자열을 포함하고 있는 선택자와 함께 모든 메서드를 찾는데 사용됩니다.

- 메서드 파인더 (the method finder) 도구는 여러분이 메서드들을 찾도록 해주며, 그 메서드들의 이름뿐만 아니라, 그 메소스들의 기능에 따라 해당 메서드를 찾게 합니다.

- 몬티첼로 브라우저 (the Monticello Browser)는 몬티첼로 패키지 (Monticello package) 로부터 코드를 로드하고, 그 패키지에 저장하기 위한 작업들의 시작점 (the starting point) 이 됩니다.

- 프로세스 브라우저 (processor browser)는 스몰토크에서 실행되는 모든 프로세스 (threads) 에 대한 뷰 (view) 를 제공합니다.

- 테스트 러너 (the test runner)는 SUnit tests를 실행하고 디버그 할 수 있게 해주며, 이 내용은 7장 에서 자세히 설명하겠습니다.

- Transcript는 로그 메시지를 작성하기에 유용한 Transcript 출력 스트림 (the Transcript output stream) 에 있는 창이며, 1.4절 에 이미 설명되어 있습니다.

- Workspace는 여러분이 직접 입력할 수 있는 창입니다. 이 창은 어떤 목적에라도 사용이 가능하지만, 가장 많이 사용되는 용도는 스몰토크표현식을 입력하고 Do IT 으로 표현식을 실행하는 것입니다. 워크스페이스의 사용 또한 1.4절 에 이미 설명했습니다.

Debugger(디버거)는 분명한 역할을 갖고 있지만, 다른 프로그래밍 언어들에서 사용하는 디버거와 비교할 때, 좀 더 핵심적인 역할을 담당하고 있음을 발견할 것입니다. 그 이유는, 여러분이 스몰토크에서, 디버거 내부에서 프로그래밍을 할 수 있기 때문입니다. 디버거는 메뉴 또는 *Tools* 플랩 으로부터 실행할 수 없으며, 실패한 테스트를 실행하거나, "CMD–." 을 입력하거나 실행중인 프로세스를 중단하거나 코드에 self halt 표현식을 삽입하여 디버거를 사용할 수 있습니다.

6.2 시스템 브라우저

스퀵에서는 사실 여러 개의 브라우저가 있습니다: 표준 시스템 브라우저(the standard system browser), 패키지 브라우저(the package browser), 옴니브라우저(the omnibrowser) 그리고 리팩토링 브라우저(Refactoring Browser) 등이 있습니다. 다른 브라우저들은 차이점들이 있기 때문에 표준 시스템 브라우저부터 살펴보겠습니다. 그림 6.1 은 *Tools* 플랩[1]에서 처음 브라우저를 외부로 드레그 하였을 때 의 모양입니다.

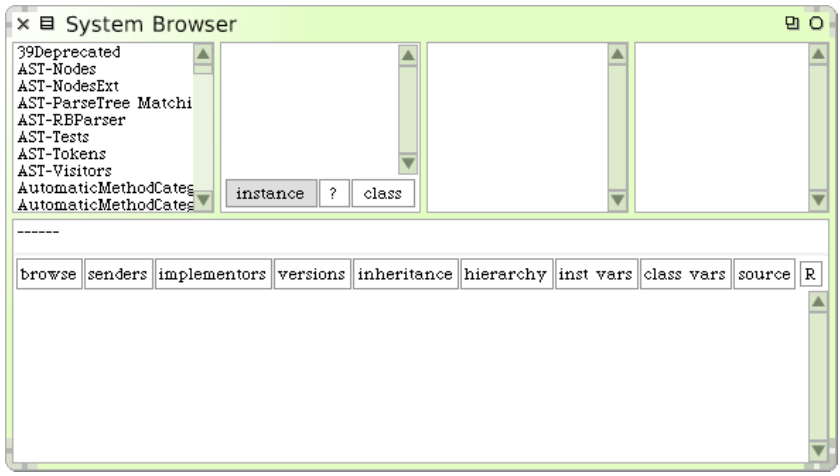


그림 6.1: 시스템 브라우저

시스템 브라우저는 NeXTSTEP과 매우 유사한 방식으로, 브라우저의 상단에 있는 4개의 패널은 시스템에서 메서드들의 계층 뷰를 나타냅니다. 컬럼모드(Column mode)에서 파일뷰어(File Viewer)와 Mac OS X의 Finder는 디스크에 있는 파일들의 뷰를 제공합니다. 가장 왼쪽의 패널은 클래스의 카테고리들을 나열하며, 카테고리들 중 하나를 선택하면, (Kernel-objects를 말합니다) 오른쪽에 있는 패널은 즉시 그 카테고리에 있는 모든 클래스들을 보여줍니다.

[1]만약 당신이 보고있는 브라우저가 우리가 설명하는 브라우저와 모양새가 다를 경우, 설치된 이미지에서 기본브라우저를 사용할 수 있을 것입니다. FAQ 5, 272페이지를 보십시오.

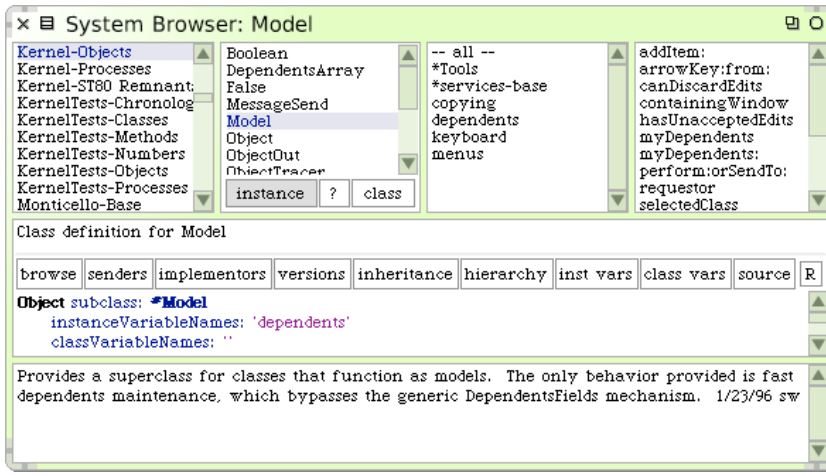


그림 6.2: `Model` 클래스를 선택한 모습의 시스템 브라우저

비슷하게, 만약 두 번째 패널에서 클래스들 중 하나를 선택하였다면, `Model` (그림 6.2 를 보세요), 세 번째 패널은, 기본으로 선택되어있는 가상 프로토콜 -- all-- 뿐만 아니라, 클래스를 위해 디자인된 모든 프로토콜을 나열할 것입니다. 프로토콜은 메서드들을 분류*categorizing* 하는 방법입니다; 프로토콜은 일관된 작은단위의 개념으로 분류를 진행 함으로서 클래스의 동작에 대해 쉽게 이해할 수 있도록 해줍니다. 네번째 창은 선택한 프로토콜에 있는 모든 메서드의 이름을 보여줍니다. 만약 메서드 이름을 선택했다면, 시스템 브라우저의 아래에 있는 큰 패널에, 사용자가 보고, 편집하고, 편집된 버전을 저장할 수 있도록 선택한 메서드의 소스코드가 나타납니다. 만약 `Model` 클래스, *dependents* 프로토콜과 `myDependents` 메서드 를 선택했다면 브라우저는 그림 6.3 과 같이 나타나게 됩니다.

Mac OS X의 Finder에서 보이는 디렉토리들과는 다르게, 브라우저의 4개의 상단 패널의 역할은 같지 않습니다. 클래스들과 메서드들이 스몰토크언어들의 일부인 반면, 시스템 카테고리들과 메시지 프로토콜들은 그렇지 않습니다:이것들은 각 패널에 봐야 할 필요가 있는 정보의 양을 제한하기 위해 브라우저에서만 사용되는 편리기능입니다. 예를 들어, 시스템 브라우저는, 만약 어떤 프로토콜도 없을 경우, 선택된 클래스에 있는 모든 메서드의 목록을 한

꺼번에 보여줘야 할 것이며, 이 목록은 많은 클래스들을 찾기 위해 편리하게 구석구석을 검색하기에는(navigation) 너무 많습니다.

이렇기때문에, 새로운 카테고리를 만들거나 새로운 프로토콜을 만드는 작업은, 새로운 클래스 또는 새로운 메서드를 만드는 작업과는 다릅니다. 새로운 카테고리를 만들려면, 카테고리 패널의 노랑 버튼 메뉴에서 `new category`를 선택하며, 프로토콜 창의 노랑버튼 메뉴에서 `new protocol`을 선택합니다.

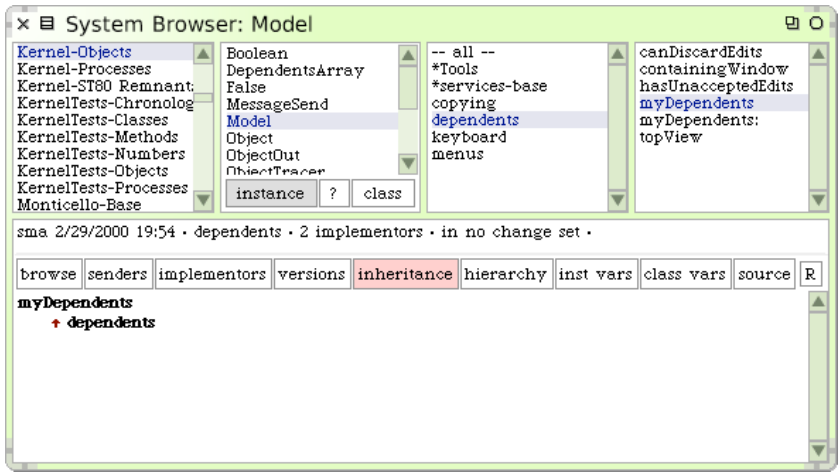


그림 6.3: 클래스 `Model`에서 `myDependents` 메서드를 보여주는 시스템 브라우저

대화 상자에 있는 새로운 요소에 이름을 입력하면 추가작업은 끝입니다: 새로운 요소의 이름과 컨텐츠들 외에 카테고리 또는 프로토콜에 더 많은 내용을 넣어야 할 것은 없습니다.

반대로, 새로운 클래스 또는 새로운 메서드를 만들려면, 스몰토크코드를 일부분이지만 실제로 작성해야 합니다. 만약 현재 선택된 카테고리(가장 왼쪽 패널) 선택을 해재하고 다시 선택한다면, 메인 브라우저 패널은 클래스 생성 템플릿(그림 6.4)을 출력합니다. 나타난 템플릿을 편집함으로써, 새로운 클래스를 만들 수 있습니다: 만들려는 하위클래스의 부모클래스가 될 클래스명으로 Object 부분을 교체하고, 만들고자하는 새로운 하위 클래스의 명칭으로 NameOfSubclass 부분을 교체한 다음, 미리 알고있는 경우, 인스턴스 변수 이름을 채워 넣습니다. 새로운 클래스를 위한 카테고리는 기본적으로 현

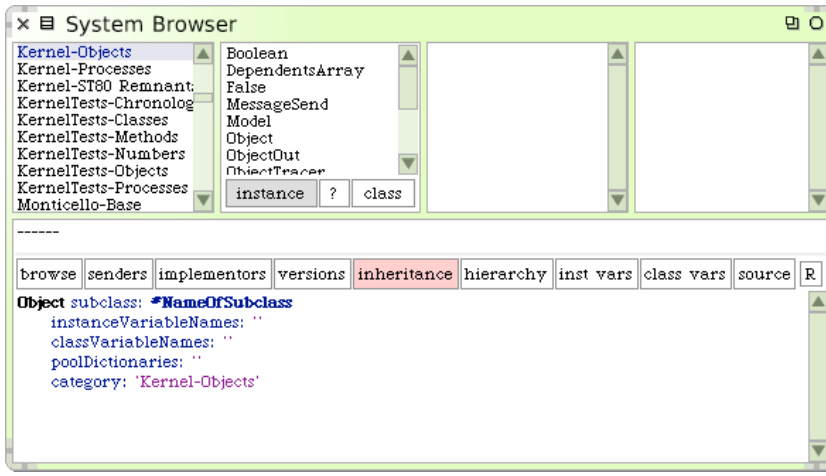


그림 6.4: 클래스 생성 탬플릿을 보여주는 시스템 브라우저

재 선택된 카테고리이지만, 원하는 경우, 이 카테고리를 변경할 수 있습니다. 당신이 하위로 분류 하려는 클래스에 이미 브라우저가 포커스되어 있다면, 클래스 패널에서 노랑 버튼 메뉴를 사용하거나 more ▷ ⋯.subclass template 를 선택하여 약간 다른 초기화로 동일한 템플릿을 얻을 수 있으며, 선택한 클래스의 하위클래스를 만들 수 있는 템플릿을 선택할 수 있습니다. 또한 클래스 이름을 새로운 것으로 변경하여, 현존하는 클래스의 정의를 편집할 수 있습니다. 대부분의 경우, 새로운 정의를 수락하면, (대응하는 메타 클래스이므로) 새로운 클래스(the #를 따르는 이름을 가진)가 만들어집니다. 클래스를 만드는 작업은, 클래스를 참조하는 글로벌 변수를 생성하게 하고, 이 작업은 클래스의 이름들을 사용하여 모든 현존하는 클래스들을 참조할 수 있게 만드는 기초작업이 됩니다.

당신은 지금 클래스 생성 템플릿에서, 새로운 클래스의 이름이 심볼 (예를 들어, # 접두어를 가진) 로서 나타나야만 하지만, 이 클래스가 만들어진 후에, 프로그래밍 코드가 식별자*identifier*(예를 들어, `the#`가 없는) 로서 클래스의 이름을 사용하여 클래스를 참조할 수 있는지에 대한 이유를 말씀하실 수 있나요?

새로운 메서드를 만드는 진행과정은 비슷합니다. 먼저, 만들게될 메서드를

포함시킬 클래스를 선택하고, 그 다음 프로토콜을 선택합니다. 브라우저는 채워넣거나 수정할 수 있는 메서드-생성 템플릿을 그림 6.5 와 같이 표시 할 것입니다.

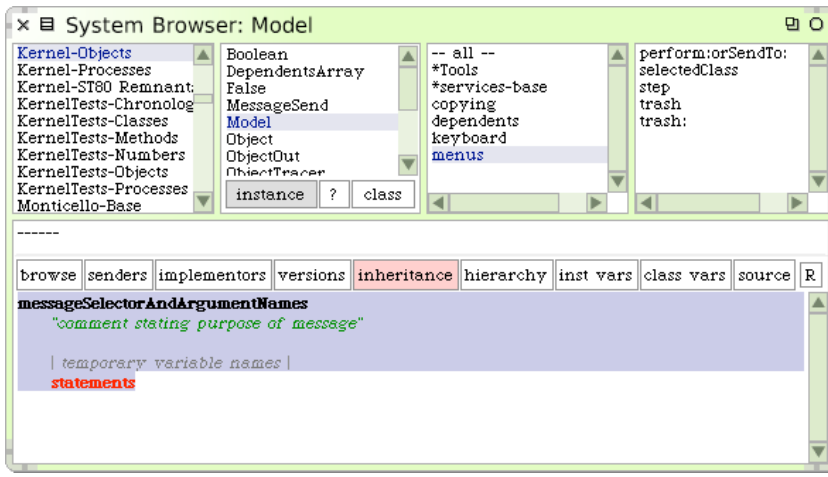


그림 6.5: 메서드 생성 탬플릿을 보여주는 시스템 브라우저

버튼 바

시스템 브라우저는 코드를 탐색하고 분석하기 위한 여러가지 도구를 제공합니다.

이러한 도구들은 브라우저 창의 중앙에 있는 수평 버튼 바에서 접근할 수 있습니다. 버튼은 완전한 세트를 보여주는 그림 6.5 와 같이 browse(검색), senders(발신자), implementers(실행자) 등의 라벨이 붙어 있습니다.

코드 탐색

Browse(검색) 버튼은, 현재 선택된 클래스 또는 메서드를 대상으로 하는 새로운 시스템 브라우저를 엽니다. 이 브라우저는 종종 동시에 여러 개의 브라우저를 열어야 하는 작업을 하는 경우에 쓸만합니다. 코드를 작성하려 한다면 적어도 다음처럼 두개 이상의 브라우저가 필요합니다:

1. 당신이 직접 입력하고 있는 메서드를 위한 브라우저
2. 입력할 내용을 보기 원하는 시스템을 검색하기 위한 브라우저.

또한 CMD–b 키보드 바로가기 메뉴를 사용하면 선택된 text의 이름으로 작성된 클래스에 대한 브라우저를 열 수 있습니다.

 이 방법을 시도해 보십시오: *Workspace* 창에서, 클래스의 이름을 입력하고*(*예를 들어, `ScaleMorph`*)*, 선택한 다음 *CMD–b* 를 누르십시오. 이 방법은 자주 유용하게 쓰이며, 모든 텍스트 창에서 작동됩니다.

메시지와 메시지의 실행

senders(발신자) 버튼은 선택된 메서드를 사용할 수 있는 모든 메서드들의 목록을 제공할 것입니다. ScaleMorph 클래스를 대상으로 정보를 보여주고 있는 브라우저에서, 브라우저의 우측상단 모서리 근처에 있는 메서드 창에 있는 checkExtent: 메서드를 클릭하면, checkExtent: 메서드의 내용이 브라우저의 아래 부분에 표시됩니다. 이 상태에서 senders 버튼을 클릭하면, 가장 상단에 현재 메서드가 나오고, 그 아래, checkExtent: 메세지를 보낼 수 있는 (그림 6.6 을 보십시오) 모든 대상의 목록이 나타납니다. 메뉴의 아이템을 선택하면 브라우저는 이미지상에 존재하는 목록상의 모든 메서드에 선택한 메세지를 전송하고 그 결과를 보여줍니다.

Implementers(구현자) 버튼은 동일한 방식으로 작동하지만, 메시지의 발신자를 목록으로 보여주는 대신, 동일한 선택자(selector)를 실행하는 모든 클래스를 목록화합니다. 이 목록을 보려면, 일단 메시지 패널에 있는 drawOn:을 선택하고, Implementers 버튼을 사용하거나, 또는 메서드 패널에 있는 노랑색 버튼 메뉴를 사용하거나, 또는 단지 선택된 drawOn:과 함께 메서드 패널에 있는 CMD–m (실행자를 위해) 단축키를 눌러서 `drawOn:`의 구현자 브라우저를 불러옵니다. 이렇게 하면 반드시, drawOn: 메서드를 실행하는 96개의 클래스가 스크롤되는 목록을 보여주는 메서드 목록 화면을 볼 수 있습니다. 너무나 많은 클래스가 이 메서드를 실행하는 현상에 전혀 놀라실 필요가 없습니다: drawOn:은 그 자신을 화면에 그릴 수 있는 모든 객체들이

이해할 수 있는 메시지입니다[2]. 이 클래스들중 하나를 보는동안, drawOn: 메시지의 발신자 검색을 시도해 보십시오. 우리는 이 메시지를 발송한 63개의 메서드를 발견하였습니다. 사용자는 또한 메시지를 선택하고 (만약 그 메시지가 키워드 메시지일 경우 인수를 포함), CMD–m 을 눌러서 해당 메서드의 구현자 브라우저를 불러올 수 있습니다.

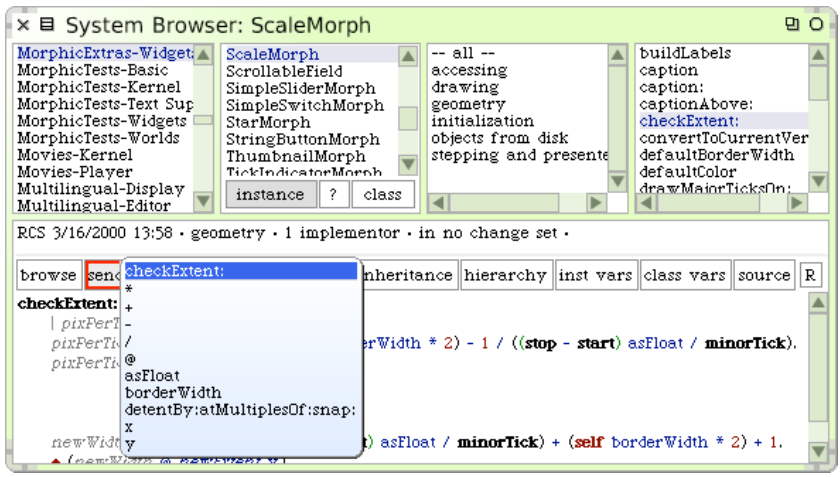


그림 6.6: ScaleMorph 클래스 위에 클래스 브라우저가 열렸습니다. 브라우저의 중앙에 버튼의 수평바가 있음에 주목해 주십시오. 이 위치 (`drawOn:`을 선택한 위치) 에서 발신자 버튼 (senders button) 을 사용할 것입니다.

만약 `drawOn: in AtomMorph>>drawOn:` 의 send 를 보았다면, 이 전송이 super send 임을 알게될겁니다. 그러므로, 우리는 실행될 메서드가 실제로는 AtomMorph의 수퍼클래스 (superclass) 에 있게 될 것을 알게됩니다. 이 클래스는 무엇일까요? hierarchy(계층) 버튼을 클릭하면 이것이 EllipseMorph 라는것을 볼 수 있습니다. 이제 그림 **??** 에서 보이는 것과 같이 `Canvas>>draw`의 목록에서 다섯번째 발신자 (sender) 를 보겠습니다. 해당 메서드는 인수로서 그 메서드에 전달된 어떤 객체에 drawOn:을 전송한다는 것을 알 수 있고, 메서드의 인수는 모든 클래스의 인스턴스가 될 수 있습니다. 이런

[2]화면상에 보이는 모든 Morph가 drawOn: 을 가지고 있다는 의미

과정을 통한 데이터 흐름 분석(Dataflow analysis)은 몇몇 메시지의 수신자(receiver)의 클래스를 알아내는 작업에 도움을 줄 수 있습니다. 하지만 일반적으로 어떤 메시지 전송(message-sends)에 의해, 메서드가 실행되는 원인이 제공되는지에 관해 브라우저가 알 수 있는 쉬운 방법은 없습니다. 그렇기 때문에, 발신자(senders) 브라우저는 이름에 대한 제안목록을 정확히 보여줍니다: 메시지의 모든 발신자(senders)는 선택한 선택자와 함께 있습니다. 그렇지만 Senders 버튼은 메서드 사용법을 숙지했을 때 매우 유용합니다: 이 버튼은 신속하게 메서드의 예제 사용법들을 찾아볼 수 있게 합니다. 비록 동일한 선택자(selector)를 가진 모든 메서드들이, 동일한 방식으로 사용되어야 하지만, 모든 주어진 메시지들의 사용은 비슷해야만 합니다.

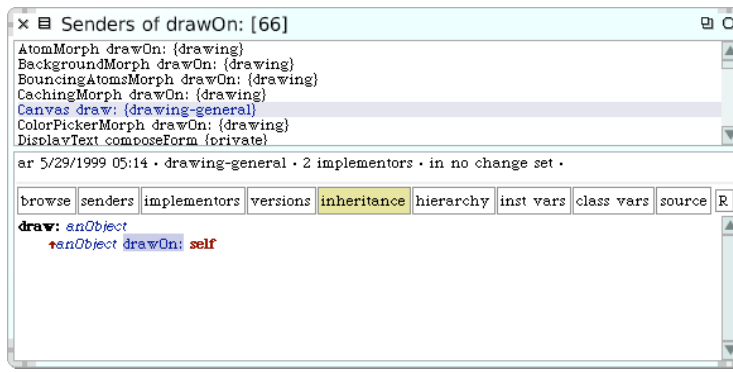


그림 6.7: 발신자 브라우저(the senders Browser)는 `Canvas>>draw` 메서드가 `drawOn:` 메시지를 그 자체의 인수에 보내는 장면을 보여줍니다.

메서드의 버전

사용자가 메서드의 새로운 버전을 저장할 때, 이전 버전을 잃어버리는것은 아닙니다. 스퀵은 모든 이전 버전들을 유지하며, 다양한 버전들을 비교할 수 있도록 해주고, 옛 버전들로 돌아갈 수 있는 복귀(revert) 기능을 제공합니다. Versions 버튼은 선택된 메서드의 정상작동하는 버전들에 대한 목록을 출력하고 관련된 작업을 할 수 있게 해줍니다. 그림 6.8 에서, 우리는 2장 장에서 설명된 Quinto 게임을 작성하는 동안 만들어진 저자들의 메서드 중 하나인

mouseUp: 메서드의 각종 버전들을 확인할 수 있습니다.

상단 패널은 메서드의 각 버전을 위해, 프로그래머 이름, 날짜 시간, 저장소, 클래스 이름, 메서드 이름, 정의된 프로토콜 장소를 나타내는 이니셜을 한줄에 표시합니다. 현재 사용되고있는 버전은 목록의 최상단에 있으며 특정 버전을 선택하면 아래 패널에 표시됩니다. 그림 6.8 에 표시된 것 처럼, diffs 체크박스가 선택되면, 화면에서 선택된 버전과 옛 버전들 사이의 차이점들을 즉시 확인할 수 있습니다. 또한 선택된 메서드와 현재 버전의 차이들을 표시하고, 선택된 버전으로 돌아가기 위한 버튼들도 있습니다. prettyDiffs 체크박스는, 레이아웃에 변경사항들이 있을 경우 유용합니다 : 이 체크박스는 변경을 가하기 전 버전들을 prettyprint하여, 포맷 변환을 제외한 차이점들이 표시되게 합니다.

버전 브라우저가 있다는건, 필요없는 코드에 대한 보관을 신경쓸 필요가 없다는 의미입니다 : 편하게 코드를 삭제하십시오. 만약 이전 코드 버전이 필요하다면, 언제라로 이전 버전으로 복귀하거나 이전 버전에 있는 코드 조각을 복사해서 다른 메서드에 복사해 넣을 수 있습니다.

버전들을 사용하는 습관을 들이세요. 더 이상 필요없는 코드를 "주석 처리" 하는 것은 좋지않은 방법입니다. 왜냐하면 사용하지 않는 코드에 대한 주석들은 현재 코드를 읽기 어렵게 만들기 때문입니다. 스몰토크사용자들은 코드의 가독성을 대단히 중요하게 생각합니다.

HINT 만약 메서드를 완전히 지웠는데, 다시 지우기 이전의 상태로 돌아가고 싶은 경우는 어떻게 하죠? 노랑 버튼 메뉴로 버전들 보기를 요청할 수 있는 장소인 변경 세트 *(a change set)* 에서 삭제된 내용을 찾을 수 있습니다. 변경 세트 브라우저는 *6.8*절 에 설명되어 있습니다.

메서드 재지정 (Method overridings)

inheritance(상속) 버튼은 표시된 메서드로 재지정 (override) 된 모든 메서드들을 보여주는 특별한 브라우저를 엽니다. 어떻게 작동되는지 보려면, `ScaleMorph>>defaultColor` 메서드를 디스플레이한 후, inheritance 를 클릭합니다. defaultColor 메서드의 정의는 그림 6.9 에서 볼 수 있듯이, `Rectan`

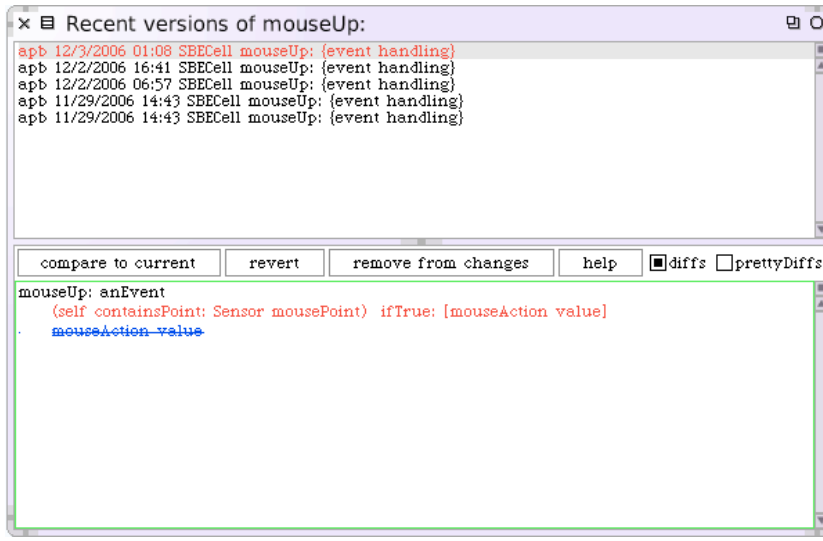


그림 6.8: 버전 브라우저 는 `SBECell>>mouseUp:` 메서드의 여러 가지 버전들을 보여줍니다.

`gleMorph>>defaultColor`를 재지정하며, 이것은 `Morph>>defaultColor`를 재지정한것입니다. Inheritance 버튼의 색상은 재지정 이 생기는 방식에 따라 틀려집니다. 색상들은 풍선도움말을 참고하세요:

분홍(pink): 보여지는 메서드는 다른 메서드를 재지정하지만, 메서드를 사용하지 않습니다.

초록(green): 보여지는 메서드는 다른 메서드를 재지정하고, super 를 통해 그 메서드를 사용합니다.

황금(gold): 보여지는 메서드 자체가 서브클래스에서 재지정됩니다.

연분홍(salmon): 보여지는 메서드가 다른 메서드를 재지정하고, 그 자신을 재지정합니다.

보라(violet): 보여지는 메서드가 다른 대상을 재지정하거나, 스스로 재지정되고, super-send를 만듭니다.

현재 두 가지 버전의 상속 브라우저(inheritance browser)가 있습니다. 만약 옴니브라우저 프레임워크(OmniBrowser framework)에 기초하여 시스템 브라우저를 사용하고 계시다면, inheritance 버튼의 색상은 변경되지

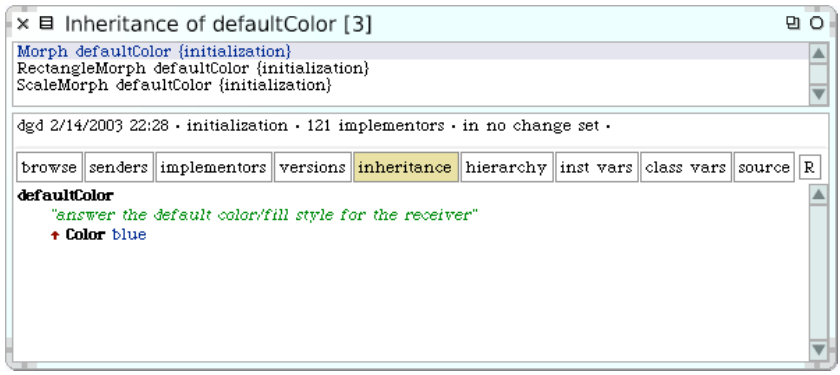


그림 6.9: 상속 순서로 본 `ScaleMorph>>defaultColor`와 이것을 재지정하는 메서드들. Inheritance 버튼은 황금색이며, 그 이유는 보여지고있는 메서드가 서브클래스에서 재지정되었기 때문입니다.

않으며, 상속 브라우저는 다른 모양이 됩니다. 이 버전은 좀더 많은 정보를 디스플레이합니다: 상속브라우저는, 상속 사슬(the inheritance chanin)에 있는 메서드를 보여드릴 뿐만 아니라, 그것들의 형제 자매들도 보여드립니다. 그림 6.10 은 상속 브라우저의 다른 버전의 모습입니다.

계층 브라우저

hierarchy 버튼은 현재 클래스에 대한 계층 브라우저 를 엽니다. 또는 이 브라우저는 클래스 창에 있는 the browse hierarchy 메뉴를 사용해서 열어도 됩니다. 계층 브라우저는 시스템 브라우저와 비슷하합니다. 하지만 맨 왼쪽에 클래스의 카테고리를 표시하는 대신, 선택한 클래스를 표시하며, 들여쓰기를 사용해서 상속된 클래스들을 출력합니다.

계층 브라우저는 상속 계층의 구석구석을 검색하는 작업을 쉽게 해주도록 고안되었지만, 시스템에 있는 모든 클래스를 보여주는건 아닙니다: 오직 초기 super클래스와 하위 클래스만을 보여드립니다. 그림 6.11 을 보면, 계층 브라우저에서 `ScaleMorph`의 super클래스는 `RectangleMorph` 임을 확인할 수 있습니다.

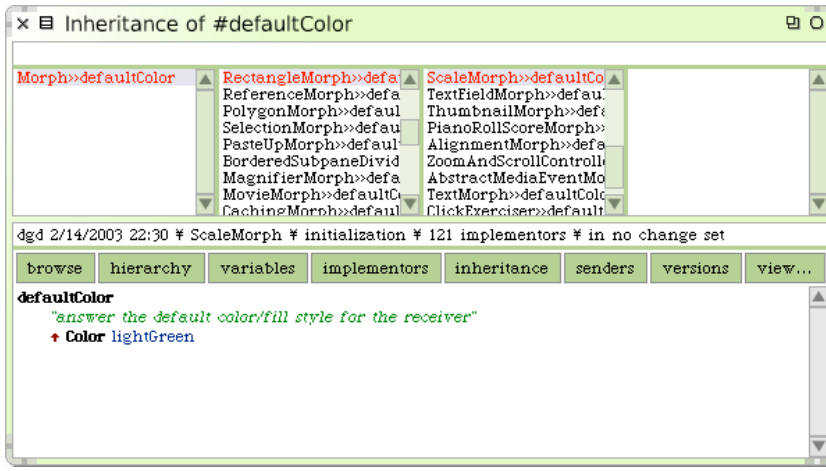


그림 6.10: 상속 브라우저에 기초하여 새로운 옴니 브라우저가 보여주듯이, `ScaleMorph>>defaultColord`와 그것이 재지정하는 메서드. 스크롤 되는 목록에 선택한 메서드의 형제(메서드)가 보입니다.

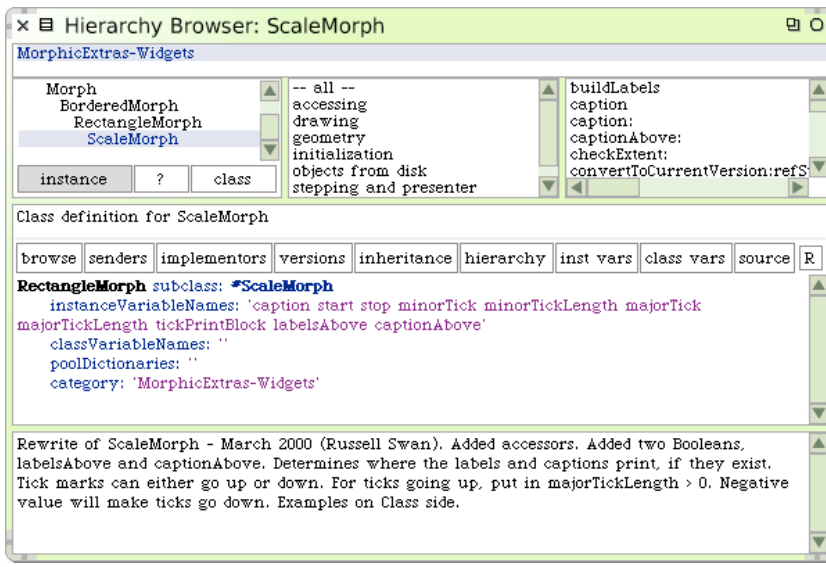


그림 6.11: `ScaleMorph` 클래스에 대한 계층브라우저

변수에 대한 참조를 찾기

Inst vars와 class vars 버튼은 어디서 인스턴스 변수와 클래스 변수가 사용되었는지를 찾는 작업을 도와줍니다; 동일한 정보에, 노랑 버튼 메뉴 inst var refs 와 클래스 패널에 있는 class var refs 을 사용할 수 있습니다. 메뉴는 또한 inst var defs 를 포함할 수 있으며, 이것은 변수에 할당된 인스턴스 변수를 참조하는것들의 서브셋을 보여줍니다. 일단 버튼을 클릭하거나 메뉴 아이템을 선택하면, 현재 클래스에서 정의된 모든 변수들과, 클래스가 상속하는 모든 변수들 중의 하나를 선택하는 대화상자를 보게됩니다. 이 목록은 상속 순서대로 되어있으며, 인스턴스 변수의 이름을 사용자에게 상기시켜 주기위해 이 목록은 자주 불러지게 됩니다. 대화상자의 바깥쪽을 클릭하면, 목록이 사라지고 어떤 변수 브라우저도 진행되지 않습니다.

클래스 패널의 class vars 버튼에서 노란버튼을 사용하면, 현재 클래스의 클래스 변수들과 그 클래스 변수들의 값을 보여주는 인스펙터가 열립니다. 그리고 class refs(N) 를 사용하면 현재 선택한 클래스 변수를 참조하는 모든 클래스 변수를 보여주게 되죠.

소스

source 버튼은 "what to show" 메뉴를 불러오며, 이 메뉴는 브라우저의 소스창에서 보여지는형태를 선택할 수 있게 해줍니다. source 코드 prettyPrinted 소스 코드, byteCodes , 그리고 바이트 코드로 부터 decompiled(디컴파일된) 소스코드도 포함됩니다. 버튼에 있는 라벨은, 다른 모드들 중 하나를 선택하면 변경됩니다. 물론 다른 옵션들도 존재하며, 만약 이름에 마우스 포인터를 올려놓으면, 풍선도움말이 나타납니다. 직접 경험해보시기 바랍니다.

"What to show" 메뉴에서 prettyPrint 를 선택하면, 코드를 저장하기 전에 메서드를 prettyPrinting 하는것과 동일한 작업이 되는건 아닙니다. 이 메뉴는 단지 브라우저가 소스를 보여주는 형태만 변경하며, 시스템에 저장된 코드에는 아무 영향을 미치지 않습니다. 두 개의 브라우저들을 열어서, 첫 번째 브라우저에서 prettyPrint 를 선택하고, 두 번째 브라우저에서 source 를 선택해서 작동상황을 확인할 수 있습니다. 사실, 동일한 메서드에서 두 개의 브라우저

에 집중하여, 한 개의 브라우저에서 byteCodes 을 선택하고 다른 브라우저에서 decompile 을 선택하는 것은, 스퀵가상 머신의 byte-coded instruction set 을 공부하는 좋은 방법입니다.

리팩토링 (Refactoring)

버튼 바 끝에 있는 작은 R[3] 버튼이 있는걸 보셨나요? 크게 눈에 띄지는 않지만, 이 버튼은 스몰토크환경에서 가장 강력하고 중요한 기능중 하나를 사용할 수 있게 합니다. R 버튼을 클릭하면, 코드에 대한 리팩토링을 할 수 있는 메뉴들을 사용이 가능합니다. 동일한 리팩토링 엔진을 여러 가지 다른 방식으로 사용할 수 있습니다. 예를 들어, 클래스, 메서드, 그리고 코드 창에서 노랑색 버튼을 통해 클래스 계층에 접근할 수 있습니다. 리팩토링은 예전에 리팩토링 브라우저라 불리는 특별한 브라우저에서만 사용할 수 있었지만, 지금은 모든 브라우저에서 리팩토링을 사용할 수 있습니다.

브라우저 메뉴

많은 추가기능들은 브라우저의 노랑색 버튼 메뉴에서 사용가능합니다. 노랑색 버튼 메뉴는 상황에 따라 반응이 틀려지기 때문에, 브라우저에 있는 각 패널은 자신만의 메뉴를 가지고 있습니다. 비록 메뉴 아이템에 있는 라벨들이 동일할 지라도, 그 라벨들의 의미는 상황에 따라 바뀝니다. 예를 들어, 카테고리 패널, 클래스 패널 프로토콜 패널과 메시지 패널은 모두 file out 메뉴 아이템을 갖고 있지만, 이 아이템들은 각각 다르게 동작합니다: 카테고리 패널의 file out 메뉴는 전체 카테고리를 file out 하며, 프로토콜 file out 의 메뉴는 전체 프로토콜을 file out 하고, 메서드 패널의 file out 메뉴는 디스플레이된 메서드만을 files-out 합니다. 비록 이런 동작이 그리 복잡한것은 아니지만, 초보자는 헷갈릴 수 있습니다.

카테고리 패널에 있는 find class… (f) 버튼은 대부분의 상황에서 유용하게 쓰입니다. 비록 카테고리들이 활발하게 개발중인 코드에 유용하다고 해도, 사실 프

[3] 당신이 만약 첫 번째로 AST를 로드하고, 그다음 스퀵소스로부터 RefactoringBrowser를 로드하고, 이후 Squeak-dev 이미지를 사용하면 이 버튼을 볼 수 있습니다.

로그래머 대부분은 전체 시스템을 카테고리화하는 작업을 잘 모르며, 클래스가 어떤 카테고리에 들어있는지 예상하는 것 보다는, CMD-f 를 이용해서 클래스 이름의 앞쪽 몇글자를 입력해서 찾는편이 훨씬 빠릅니다. Recent classes…(r) 는 사용자가 원하는 클래스의 이름을 기억하지 못해도 최근에 검색한 클래스를 이용해서 원하는 클래스로 빨리 돌아갈 수 있게 합니다.

클래스 패널에는, find method 와 find method wildcard… 의 두가지 메뉴 항목이 있으며, 이 기능은 특정한 메서드를 검색하고 싶을때 유유요합니다. 그렇지만, 메서드의 목록이 매우 긴게 아니라면, --all-- 프로토콜을 검색하고 (이것이 기본값), 마우스를 메서드 패널에 놓고, 찾고싶은 메서드 이름의 첫 번째 글자를 입력하는게 훨씬 간편합니다. 두가지 기능을 이용한다면 일반적인 경우 패널을 스크롤하여, 보이는 메서드 이름을 찾게 됩니다.

`OrderedCollection>>removeAt:` 메서드를 찾는 두 가지 방법을 시도해 보세요.

사용할 수 있는 더 많은 옵션이 메뉴에 있습니다. 이러한 옵션들은 브라우저에 대해 작동되며 결과를 확인하는데 시간이 약간 걸립니다.

클래스 창 *(pane)* 메뉴에 있는 Browse Protocol(프로토콜 탐색), Browse Hierarchy(계층 탐색) 와 Show Hierarchy(계층 보기) 의 결과를 비교해 보시기 바랍니다

다른 클래스 브라우저

7장 의 도입부분에서, 다른 클래스 브라우저인 *package pane browser* 를 언급한적이 있습니다. 이 브라우저는 월드 메뉴 (the world menu): World ▷ open… ▷ package pane browser 를 이용해서 사용할 수 있습니다. 이것은 기본적으로 클래스 브라우저와 동일하지만, 시스템 카테고리를 위한 명명규칙 관례를 알고 있습니다. 예를 들어, ScaleMorph 클래스는 Morphic-Widgets 카테고리에 속해 있습니다. 패키지 브라우저는 하이픈 전의, 일부분으로 가정하고, Morphic은 "패키지" 의 이름이며, 다섯개의 패널을 추가하여 특정한 패키지에 있는 카테고리들을 검색할 수 있도록 해줍니다. 그렇지

만, 아무 패키지도 선택하지 않았다면, 마치 보통의 4개 패널 브라우저처럼, 모든 카테고리들이 사용 가능한 상태로 됩니다.

안타깝게도, 패키지 라는 용어의 의미는 *package pane browser* 가 개발된 이후로 바뀌었습니다. 현재에서 "패키지(package)" 라고하면 다음 장에서 논의할 내용인 몬티첼로 패키징 도구와 관련된, 좀 더 정확한 의미가 있습니다. 현재, 몬티첼로로 정의된 패키지들을 검색할 수 있는 도구는 없지만, 현재 개발되고 있는 중입니다.

스퀵커뮤니티는 옴니브라우저라 불리는 새롭고, 고도로 사용자화었으며, 프레임워크에 기초한 완전히 새로운 브라우저들의 집합체를 개발하는 과정에 있습니다. 옴니브라우저의 실행은 객체 지향 디자인의 좋은 예로서 살펴볼만한 가치가 있지만, 밖에서 보면 대부분의 옴니브라우저 기반 도구들은 우리가 방금 묘사한 도구들과 매우 유사합니다. 옴니 시스템 브라우저에서 알수있는 될 주요 개량점은 virtual 프로토콜에 의한 결과 입니다. 기존의 오래된 프로그래머 정의 프로토콜 뿐만 아니라, 각 클래스는 정의된 규칙에 정해진 여러 개의 가상 프로토콜을 갖고 있습니다. 예를 들어, --required-- 프로토콜은 현재 클래스 또는 정의되지 않은 그 클래스의 super클래스에 있는 메서드들에 의해 전송되는 모든 메시지들을 나열하는 반면, --supersend-- 프로토콜은 super에 전하는 모든 메서드들을 포함합니다.

프로그램적으로 검색하기

SystemNavigation 클래스는 시스템을 자세히 볼 수 있는 몇가지 유용한 유틸리티 메서드를 제공합니다. 클래식 브라우저로 제공되는 많은 기능들은 SystemNavigation 으로 실행할 수 있습니다.

 *Workspace* 를 열고, `checkExtent:` 의 발신자*(the senders)* 를 검색하기 위해 다음 코드를 do it 하십시오.

```
SystemNavigation default browseAllCallsOn: #checkExtent: .
```

특정한 클래스 메서드들에 대한 발신자, 검색을 제한하려면 다음과 같이 하면 됩니다:

```
SystemNavigation default browseAllCallsOn: #drawOn: from:
    ScaleMorph .
```

개발도구들이 모두 객체 이기 때문에, 프로그램을 이용해서 완벽하게 접근할 수 있으며, 당신은 자신의 도구들을 개발하거나, 필요에 따라 이미 만들어져 있는 도구들을 선택할 수 있습니다.

implementers 버튼과 같은 동작을 하는 프로그램 코드는 아래와 같습니다:

```
SystemNavigation default browseAllImplementorsOf: #checkExtent: .
```

어떤것을 할 수 있는지 더 공부하려면, 시스템 브라우저를 사용하여 ystem-Navigation 클래스를 탐색합니다.

더 많은 탐색 예는, 부록 12 에서 볼 수 있습니다.

요약

살펴봤듯이, 스몰토크코드를 탐색하는 데는 여러가지 방법이 있습니다. 처음에는 헷갈릴 수 있겠지만, 그러한 경우에는 항상 전통적인 시스템 브라우저로 복귀할 수 있습니다. 이렇게 복잡할 수 있음에도 불구하고, 초보자들이 스퀵에 익숙해 짐에 따라, 다양한 브라우저들이, 오히려 장점들중 하나라는것을 알 수 있는데, 이러한 브라우저들 덕분에 코드를 이해하고 만드는 작업에 대한 많은 기능을 제공받기 때문입니다. 코드 독해의 문제는 대용량 프로그램 개발에 있어 큰 도전과제중 하나입니다.

6.3 몬티첼로 (Monticello)

2.9절 에서 스퀵의 패키징 도구인 몬티첼로 에 대해 간단하게 알아봤습니다. 하지만, 몬티첼로에는 알려드린 내용보다 더 많은 기능이 있습니다. 이는 몬티첼로가 패키지를 관리하기 때문입니다. 그렇지만 이 절 에서는 몬티첼로에 관해 말씀드리기 전에 알아야할 중요한 내용인, 패키지 는 정확히 무엇인지 부터 설명하도록 하겠습니다.

패키지 : 스퀵코드의 선언적 분류 (declarative categorization)

패키지 시스템은 단순하며, 스몰토크소스 코드의 체계를 구성하는 가볍고 간략한 방법입니다. 패키지 시스템은 덧붙여 ??절 에서 살펴본 *오래된 작명관례* 라고 불리는 방법을 사용합니다.

예를 들어, 설명해 보겠습니다. 스퀵에서 관계형 데이터 베이스를 사용하기위해 개발하고 있는 프레임 워크의 이름을 짓는다고 가정해 봅시다. 만들어지는 프레임워크를 SqueakLink라고 부르고, SqueakLink 프레임워크에 작성된 모든 클래스들을 포함하는 일련의 시스템 카테고리를 만들어 보겠습니다.

```
Category 'SqueakLink-Connections' contains OracleConnection MySQLCo
nnection PostgresConnection
Category 'SqueakLink-Model' contains DBTable DBRow DBQuery
```

위에서 보이는것처럼 여러개의 카테고리와 클래스등등이 만들어지겠죠. 하지만 저 클래스들 모두에 일일히 코드를 넣어줘야 하는건 아닙니다. 예를 들자면, SQL 친화적인 상태로 오브젝트를 변환해주는 메소드 등이 있을 수 있습니다.

```
Object>>asSQL
String>>asSQL
Date>>asSQL
```

위의 메서드들은 `SqueakLink-Connections` 카테고리와 `SqueakLink-Model` 카테고리에 소속되게 됩니다. 하지만 분명한건 당신이 만들게 되는 모든 패키지에 Object 클래스가 소속되어 있는건 아니라는겁니다. 그렇기 때문에 클래스가 다른 패키지에 있는경우 패키지에 특정 메소드를 밀어넣는 방법이 필요한겁니다.

패키지에 다른 클래스의 메소드를 추가하는 작업을 한다면, *squeaklink 로 이름지어진 (초기 별표와 소문자 이름에 주의합니다) 프로토콜에 다른 객체 (`Object`, `String`, `Date` 클래스등) 의 메소드를 배치하는 작업을 해야합니다. SqueakLink 라는 패키지명은 SqueakLink-… 카테고리와 *squeaklink 프로토콜의 조합으로 만들어졌습니다. 좀 더 정확히 보자면 패키지에 대한 이름규칙은 다음과 같습니다.

Foo 패키지 이름은 아래의 규칙이 포함됩니다:

1. Foo 카테고리에 소속되어 정의된 모든 클래스, 또는 카테고리 안에서 Foo- 라는 이름으로 시작하는 클래스.
2. *foo 라는 이름의 프로토콜로 메소드를 정의하고 분류한 모든 클래스, 또는 *foo- 로 시작되는 모든 이름(이 경우 이름을 비교할때 대소문자는 무시됩니다)
3. Foo 카테고리 안의 클래스의 모든 메서드, 또는 카테고리 안에서 Foo- 로 시작되는 모든 이름. * 로 시작되는 이름의 프로토콜에 속한 메서드는 제외.

이런 규칙들때문에, 각 클래스 정의와 각 메소스들의 정의는 정확히 하나의 패키지에 속하게 됩니다. 마지막 규칙의 *에 대한 예외는 반드시 적용되어야 하며, 그렇게 해야 해당되는 메서드들이 반드시 다른 패키지에 소속될 수 있기 때문입니다. 두번째 규칙 에서 대소문자 구별을 무시하는 이유는, 관례적으로 카테고리 이름들이 CamelCase(그리고 Space문자를 포함하지 않음)를 사용하는 반면, 프로토콜 이름들은 (Space문자를 포함할 수 있음) 모두 소문자이기 때문입니다.

PackageInfo 클래스는 이런 규칙들을 적용하며, PackageInfo 클래스로 실행해서 규칙들에 대한 감을 익힐 수 있습니다.

 이 작업을 당신의 이미지에 적용하려면, *PackageInfo* 클래스와 *RefactoringBrowser*가 이미지에 포함되어야 합니다.

리팩토링 브라우저 코드(the refactoring browser code)는 패키지의 이름으로서, RefactoringEngine 과 함께 패키지 작명 관례(package naming convetions)를 사용합니다. 워크스페이스에서, 아래의 구문으로 패키지의 모델을 만들어봅시다.

```
refactory := PackageInfo named: 'RefactoringEngine'.
```

이제 refactory 를 이용해서 패키지를 자세히 살펴보는 (introspect) 작업이 가능해졌습니다. 예를 들어, refactory 클래스는 리팩토링 엔진 (Refactoring Engine) 과 리팩토링 브라우저 (Refactoring Browser) 를 만드는 긴 클래스의 목록을 반환할 것입니다. refactory coreMethods는 그 클래스들에 있는 모든 메서드들을 위한 MethodReferences 목록을 반환할 것입니다. refactoryextensionMethods는 아마도, 가장 흥미로운 질의중 하나일겁니다: refactoryextensionMethods는 RefactoringEngine에 포함되었지만 RefactoringEngine 클래스 내부에는 포함되지 않은, 모든 메서드들의 목록을 반환합니다. 예를 들어, 이 메서드들의 목록은 `ClassDescription>>chooseThisClassInstVarThenDo:` 그리고 SharedPool `class>>keys`를 포함합니다.

패키지 기능은 스퀵에 비교적 새롭게 추가된것이지만, 패키지 작명 관례 (the package naming convettions) 는 이미 사용되던 규칙을 기반으로 하기 때문에, 명확하지 않은 오래된 코드의 분석작업에 PackageInfo 를 사용할 수 있습니다.

 `(PackageInfo named: 'Collections') externalSubclasses`를 분석합니다, 이 표현식은 `Collections` 패키지에 없는 *Collection* 의 모든 하위클래스들의 목록을 답으로 내놓을 것입니다.

또한 전체 패키지의 변경 세트 (change set) 를 얻기 위해 PackageInfo의 인스턴스에 fileout 을 전송할 수 있습니다. 패키지들의 좀더 세련된 버전화를 위해, 우리는 몬티첼로 (Monticello) 를 사용합니다.

몬티첼로 기본사항 (Basic Monticello)

몬티첼로는 미국의 3 번째 대통령 토머스 제퍼슨의 산 꼭대기 집과 (the mountaintop home) 과 종교의 자유를 위한 버지니아의 법령의 저자의 이름을 본딴 것입니다. 몬티첼로는 이탈리아어로 "작은 산" 을 의미하고, 항상 이탈리어로 "c" 로 발음되며, 이 음절은 `chiair:Mont-y'-che-llo`에서 "ch" 와 같습니다.

그림 6.12 에서 볼 수 있듯이, 몬티첼로 브라우저를 열면, 두 개의 목록 패널들을 볼 수 있으며, 나열된 버튼들을 볼 수 있습니다. 왼쪽의 패널은 현재 실행

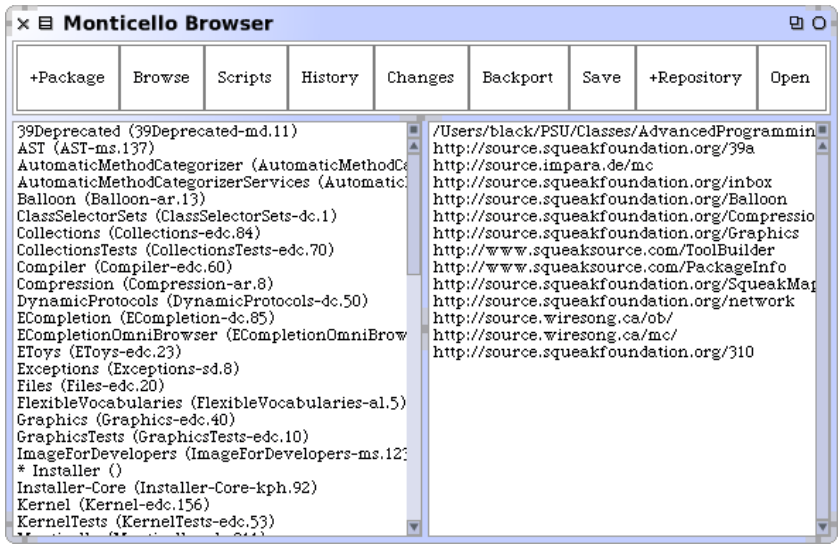


그림 6.12: 몬티첼로 브라우저 (Monticello browser)

하고 있는 이미지에 로드된 모든 패키지의 목록이며 패키지의 특별한 버전은 이름 뒤에 있는 괄호에서 확인할 수 있습니다.

오른쪽의 패널은 몬티첼로가 알고 있는 모든 소스코드 저장소들의 목록이며, 그 이유는 대개, 해당되는 저장소들로부터 코드를 로드했기 때문입니다. 만약 왼쪽 패널에서 패키지를 선택한다면, 오른쪽 패널은 필터링을 거쳐 선택된 패키지의 버전을 포함하고 있는 저장소들만 보여줍니다.

저장소목록중 *package-cache* 라는 특정 이름의 디렉토리는, 현재 이미지가 실행중인 디렉토리의 하위 디렉토리입니다. 원격 저장소 (remote repository) 로부터 코드를 로드 하거나 작성할 때, 복사본이 패키지 캐쉬에 저장되며, 만약 네트워크가 사용 가능하지 않을 때나 패키지에 접근할 필요가 있을때 유용합니다. 그리고, 만약 이메일 첨부파일로 직접 몬티첼로 (.mcz) 파일을 받으셨다면, 다운받은 파일을 사용할때 접근하기 위한 가장 편리한 방법은 package-cache 디렉토리에 다운받은 파일을 배치하는 것입니다.

목록에 새로운 저장소를 추가하려면 +Repository 를 클릭하고, 팝업 메뉴에

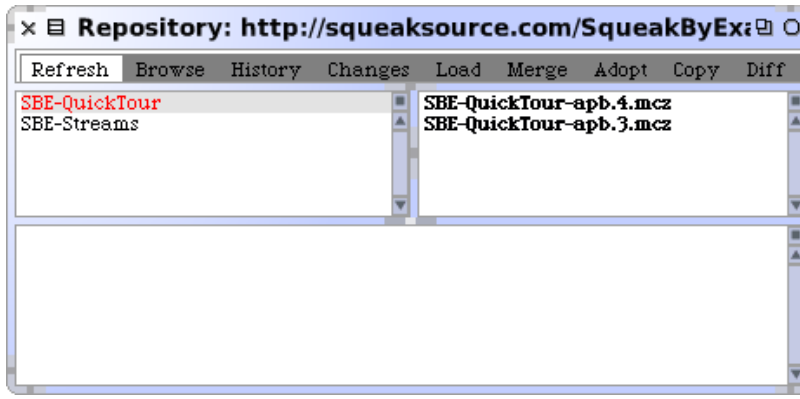


그림 6.13: 저장소 브라우저 (A Repository browser)

서 저장소의 종류를 선택합니다. `HTTP` 저장소를 추가하는 과정을 진행해보겠습니다.

몬티첼로를 열고 +Repository 버튼을 누르고 `HTTP`를 선택합니다. *Dialog*를 읽고 편집합니다.

```
MCHttpRepository
    location: 'http://squeaksource.com/SqueakByExample'
    user: ''
    password: ''
```

그 다음 선택한 저장소에 대한 저장소 브라우저 (repository browser)를 열기 위해 Open을 클릭합니다. 그림 6.13 과 같은 모습이 보일겁니다. 왼쪽은, 대상이 되는 저장소에 있는 모든 패키지의 목록이며, 만약 패키지중 하나를 선택하면, 오른쪽에 있는 패널에는 선택 패키지에 대한 이 저장소에 있는 모든 버전을 보여줍니다.

만약 왼쪽목록에 나온 버전중 하나를 선택하면, 선택한 버전을 (현재의 이미지에 로드하지 않고) Browse (검색) 하실 수 있으며, 현재의 이미지로 Load 함으로써 사용자의 이미지에 발생한 Changes 들을 볼 수 있습니다. 또한 패키지의 버전에 대한 Copy 상태를 만들고, 다른 저장소에 저장할 수도 있습니다.

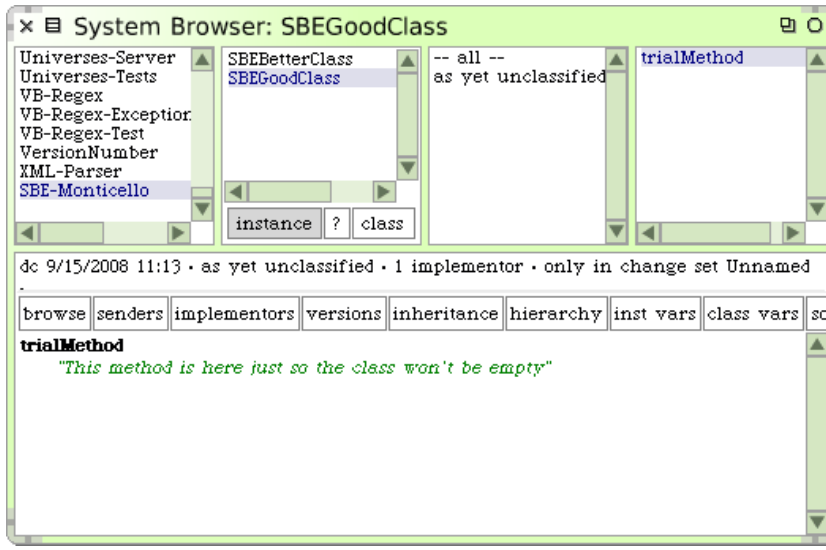


그림 6.14: "SBE" 패키지들에 있는 두 개의 클래스들

이렇듯이, 패키지의 이름을 포함하고 있는 버전의 이름은, 버전의 저자와 버전 번호의 머릿글자입니다. 버전 이름은 또한 저장소에 있는 파일 이름이기도 합니다. 이 이름들을 절대로 변경하시지 마세요. 몬티첼로의 수정 작업은 이 이름들을 기초로 진행됩니다. 몬티첼로 버전 파일들은 단지 Zip archives 입니다. 그리고 궁금하다는 이유로 zip 도구를 이용해서 파일 압축을 해제할 수도 있지만, 파일의 내부를 살펴보는 제일 좋은 방법은 몬티첼로를 사용하는 것입니다.

몬티첼로로 패키지를 만들려면, 두 가지 작업을 하셔야 합니다: 약간의 코드 작성, 그리고 그것에 대해 몬티첼로에게 알려줘야 합니다.

`SBE-Monticello`라는 이름의 카테고리를 만들고, 그림 *6.14* 에 보이듯이 몇 개의 클래스를 만든 카테고리에 추가합니다. 기존에 있는 클래스에 메서드를 만들고, 그림 *6.15* 에서 볼 수 있듯이 이름짓기규칙등을 사용해서 만들어진 메서드를 새로만든 패키지에 추가시킵니다

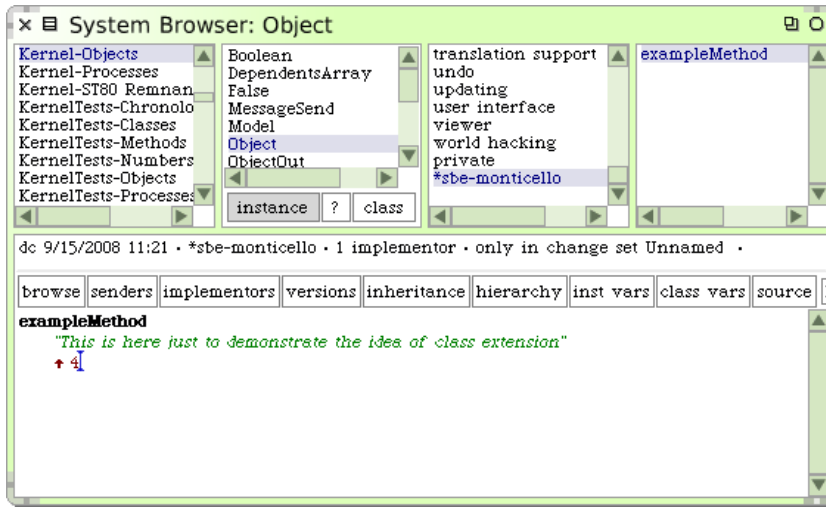


그림 6.15: "SBE" 패키지에 있게 될 확장 메서드(extension method)

몬티첼로에게 이렇게 만들어진 패키지에 관해 알려주려면, +Package 를 클릭하고, 추가를 원하는 패키지의 이름을 입력 합니다. 이 경우는 "SBE" 를 입력하도록 하겠습니다. 몬티첼로는 SBE를 몬티첼로 내부의 패키지 목록에 추가합니다. 패키지 엔트리는 이미지에 있는 버전이 어떤 저장소에도 기록되지 않았다는 것을 알려주기 위해 *(별표)로 해당 항목을 표시합니다.

저장된 결과가 없는 처음에는, 그림 6.16 에서 보이는 것 처럼, 저장소 목록에는 package-cache 하나만이 현재 패키지와 관련된 저장소로 나타나게 됩니다. 현재 상황은 우무문제 없습니다: 우리는 여전히 코드를 저장할 수 있으며, 저장작업은 코드가 package-cache에 기록되도록 해줍니다. Save를 클릭하면, 그림 6.17 에서 보이는 것처럼 잠시후 저장될 패키지의 버전에 대한 로그 메시지 입력을 요구합니다. 메시지를 입력하고 Accept 하면, 몬티첼로는 패키지를 저장합니다. 작업이 진행됐음을 나타내기 위해 몬티첼로 패키지 패널에 있는 이름에있는 별표가 제거될 것이며 새로 버전이 추가됩니다.

만약 패키지에 변경사항이 생겼다면(클래스 중의 하나에 메서드를 추가하라는등의 작업) *(별표)는 패키지가 저장되지 않은 변경사항이 있다는 것을 알

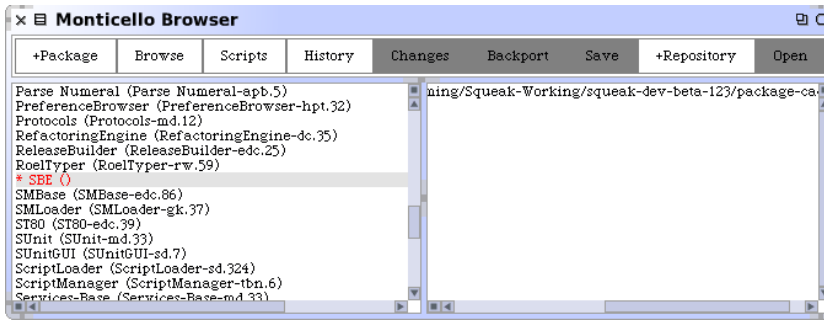


그림 6.16: 몬티첼로에 있는 아직까지 저장되지 않은 SBE 패키지

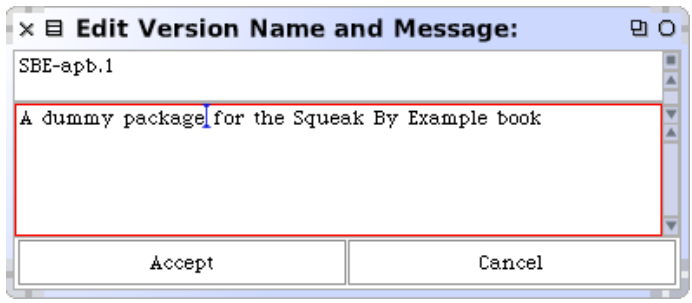


그림 6.17: 패키지의 새로운 버전을 위한 로그 메시지 입력하기

려주며 * 표가 다시 나타납니다. 만약 package-cache에 있는 저장소 브라우저를 열면, 이미 저장되었던 버전을 선택할 수 있고 Changes 들과 다른 버튼들을 사용할 수 있습니다. 물론 저장소에 새로운 버전을 저장할 수도 있습니다. 일단 저장소 뷰(the repository view)를 Refresh 하셨다면, 그림 6.18과 같이 될것입니다.

package-cache가 아닌 다른저장소에 새로운 패키지를 추가하려면, 먼저 몬티첼로가 추가하려는 저장소에 관하여 알고있도록 만들어야 합니다.

그 다음 package-cache 저장소 브라우저에 있는 Copy 를 사용할 수 있으며, 패키지의 복사대상이될 저장소를 선택하실 수 있습니다. 또한 그림 **??** 에 보이는 것처럼 노랑버튼 메뉴 아이템 `add to package` 를 사용하여 패키지와

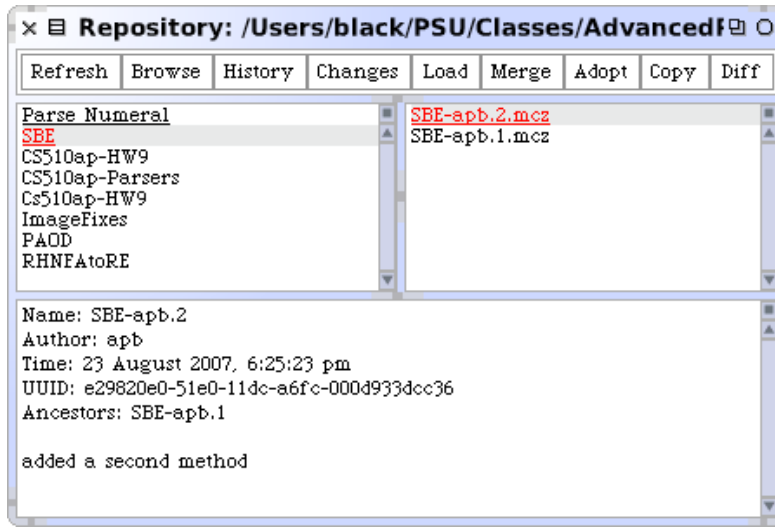


그림 6.18: 현재 package-cache에 있는 선택한 패키지의 두 개의 버전

원하는 저장소를 조합할 수 있습니다. 일단 패키지가 저장소에 관해서 알게 되면, 몬티첼로 브라우저에 저장소와 패키지를 선택하고 Save 를 클릭하여 새로운 버전을 저장하면 됩니다.

물론, 당신은 SqueakSource에 있는 SqueakByExample의저장소인 world 저장소에 대해 읽기*read* 는 가능하지만 , world저장소에 쓰기*write* 는 불가능하기때문에, 만약 world저장소에 쓰기를 시도하고 저장하면, 반드시 에러메시지가 발생합니다. 그럼에도 불구하고, www.squeaksource.com에 있는 웹 인터페이스*the web interface* 를 사용하여 사용자 자신만의 저장소를 만들 수 있으며, 당신의 작업을 저장하기 위해 이 웹 인터페이스를 사용할 수 있습니다. 이 웹 인터페이스는 특별히, 만약 여러 개의 컴퓨터를 사용하는 경우, 친구와 당신의 코드를 공유하기 위한 메커니즘으로 매우 유용합니다. 만약 쓰기 권한을 갖고 있지 않은 저장소에 쓰기 및 저장을 시도해도, 서버에는 저장되지 않으며 저장하려는 버전은 package-cache에 기록됩니다.

쓰기 권한이 없는경우 저장소에 저장을 할 수 없기때문에, 필요한경우 기존의 저장소 정보를 편집(몬티첼로 브라우저에서 노랑 버튼)하거나 또는 다른 저

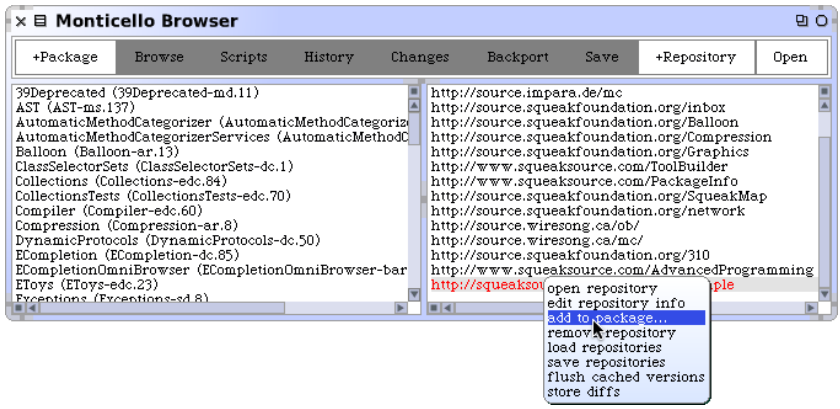


그림 6.19: 패키지와 조합된 저장소들의 세트에 저장소 추가하기

장소를 선택하고, 그 다음 패키지 캐쉬 브라우저(package-cache browser)로 부터 Copy 를 사용하여 내용을 복구(recover)하여야 합니다.

6.4 Inspector와 탐색기

다른프로그래밍 환경과 스몰토크의 차이점중 하나는, 스몰토크는 당신을 정적 코드의 세계가 아니라, window를 제공하는 객체의 세계로 초대합니다. 이렇게 제공되는 객체들은 전부 다 프로그래머가 검사하고 바꿀 수 있습니다-비록 시스템에 대한 기본 객체를 변경할 때는 주의점이 있지만요. 어떤 실험을 해도 상관없습니다만, 실험전에는 현재 이미지를 먼저 저장하는걸 잊지마세요!

Inspector

Inspector 로 할 수 있는걸 보겠습니다. *Workspace*에 `TimeStamp now` 를 입력하고, 노랑 버튼 메뉴를 사용해서 inspect it 을 클릭하세요.

(메뉴를 사용하기 전에 일일히 텍스트를 선택할 필요는 없습니다. 만약 어떤 텍스트도 선택하지 않았다면, 메뉴는 현재 라인의 전체를 대상으로 동작합니

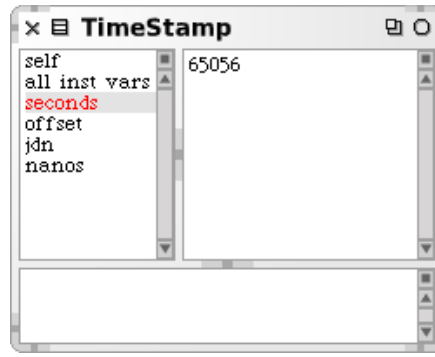


그림 6.20: `TimeStamp now`를 정밀 검사하기(inspecting)

다. 또한 inspect it 을 하기 위해 단축키로 CMD−i 를 사용할 수 있습니다.)

그림 6.20 처럼 생긴 창이 나타납니다. 이것을 Inspector라고 하며, 특정 객체의 내부에 존재하는 windows로 생각해도 됩니다. Inspector에서 참조하는 인스턴스는 TimeStamp now 프로그램식을 실행하는 시점에서 만들어지는 바로 그 instance가 됩니다.

창의 제목 표시줄은 정밀 검사할 객체의 클래스를 표시합니다. 만약 왼쪽 상단패널의 위에 있는 self 를 선택하면, 우측 패널은 객체의 printstring이 표시됩니다. 왼쪽 패널에서 all inst vars 를 선택한다면, 우측 패널은 객체에 있는 인스턴스 변수의 목록을 출력하고 각 값을 string으로 출력합니다. 그외 왼쪽 패널의 아이템들은 인스턴스 변수들을 나타냅니다. 이것은 한번에 변수들을 한번에 하나씩 검사와 변경을 진행하는 경우 유용하게 쓰입니다.

Inspector 아래에 있는 수평 패널은 작은 Workspace 창입니다. 처리한 객체에 묶입니다. 이 창에서 의사변수 self는 객체에대한 값을 정확하게 똑같이 가지고있기때문에 유용하게 쓰일 수 있습니다. 아래의 문장인

```
self -- TimeStamp today
```

를 작은 Workspace에서 inspect it 하는경우, TimeStamp now의 실행으로 인해 TimeStampe 인스턴트 객체가 생성된 시점의 시간과 자정의 시간차를 Duraion 객체(window) 로 볼 수 있습니다. 또한 `TimeStamp now - self` 를

실행해 볼 수 있으며, 이 작업은 이 책의 이 장을 얼마나 오래 읽었는지에 대한 결과[4]를 알려줍니다.

`self` 뿐만 아니라 객체의 모든 인스턴스 변수들은 작은 Workspace 패널의 영역에 있으므로, 프로그램식으로 왼쪽패널의 변수들을 사용할 수 있고, 그 변수에 값을 할당할 수도 있습니다. 예를 들어, 만약 작은 Workspace 패널에서 `jdn := jdn - 1`를 실행한다면, `jdn` 인스턴스 변수의 값이 실제로 변경되는걸 확인할 수 있고, `TimeStamp now - self`의 값이 실행할때마다 커지는걸 확인할 수도 있습니다.

여러분은 인스턴스 변수들을 선택하고, 스퀵표현식과 Accept를 사용해서, 오른쪽 패널에 있는 이전값을 교체하여 인스턴스 변수들을 직접 변경할 수 있습니다. 스퀵은 프로그램식을 실행하고 결과를 그 인스턴스에 다시 할당합니다.

이런 특별한 객체의 컨텐츠들을 쉽게 점검할 수 있도록 해주는 Dictionary와 OrderedCollections, CompiledMethods와 몇몇 다른 클래스들이 있습니다.

오브젝트 탐색기-The Object Explorer

오브젝트 탐색기는 개념적으로 Inspector와 비슷하지만, 가지고있는 정보를 알려주는 방식이 틀립니다. 차이점들을 보기 위해, 우리가 지금 정밀검사중인 동일한 객체를 탐색할 것입니다.

 왼쪽 패널에서 self를 선택하고, 노랑색 버튼 메뉴에서 explore (I)를 고르십시오.

오브젝트 탐색기는 그림 6.21 에 나온 모습과 같습니다. 만약 root 옆에 있는 삼각형을 클릭한다면, 생김새는 그림 6.22 처럼 바뀔 것이며, 아래쪽에 탐색하고 있는 객체의 인스턴스 변수들을 보여주게 됩니다. 오브젝트 탐색기는,

[4] TimeStamp now - self의 inspect 결과는 원래 TimeStamp now의 inspect창이 생성된 시점부터 TimeStamp now - self 명령어를 다시 실행한 순간까지의 시간을 의미합니다

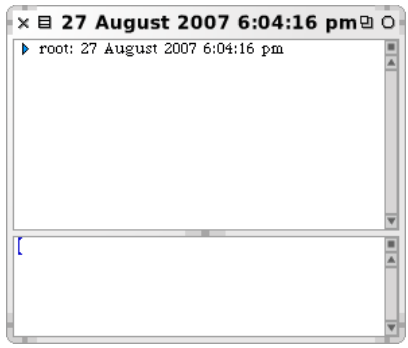


그림 6.21: Exploring `TimeStampNow` 탐색하기

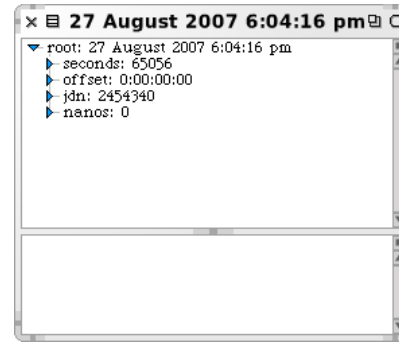


그림 6.22: 인스턴스 변수 탐색하기

복잡한 계층적 구조를 찾아볼 필요가 있을 때, 매우 유용합니다-그래서 오브젝트 탐색기 라는이름을 가지게 되었죠.

오브젝트 익스플로러 하단부의 Workspace 패널은 Inspector의 그것과는 다르게 동작합니다. self 는 root 객체 (구문이 실행되며 생성된 인스턴스) 에 묶이지 않고 오히려 현재 탐색을 위해 선택된 객체를 가리키며, 선택된 객체의 인스턴스 변수들 또한 Workspace의 범위가 됩니다.

오브젝트 탐색기의 가치를 알아보기 위해, 깊게 상주된 객체의 구조를 탐색하는 작업에 오브젝트 탐색기를 사용해보겠습니다.

메시지들의 목록을 나타내기 위해 사용되는 `PluggableListMorph`에 있는 Morphic halo를 불러오기 위해 브라우저를 열고, 메서드 패널에서 파랑색-클릭을 몇번해서 메서드의 목록부분이 선택되는 상태가 되도록 합니다.

디버그 핸들을 클릭하고, 나타나는 메뉴에서 explore Morph 를 선택합니다. 이 작업은 화면에 있는 메서드 목록을 나타내는 `PluggableListMorph` 를 탐색하는 오브젝트 탐색기를 열 것입니다. root 객체를 열고 (root옆의 삼각형을 클릭해서), 그 객체의 하위 모프를 연 다음, 그림 6.23 에 나온 것과 같이 이 모프를 기반으로 하는 객체의 구조를 탐색합니다.

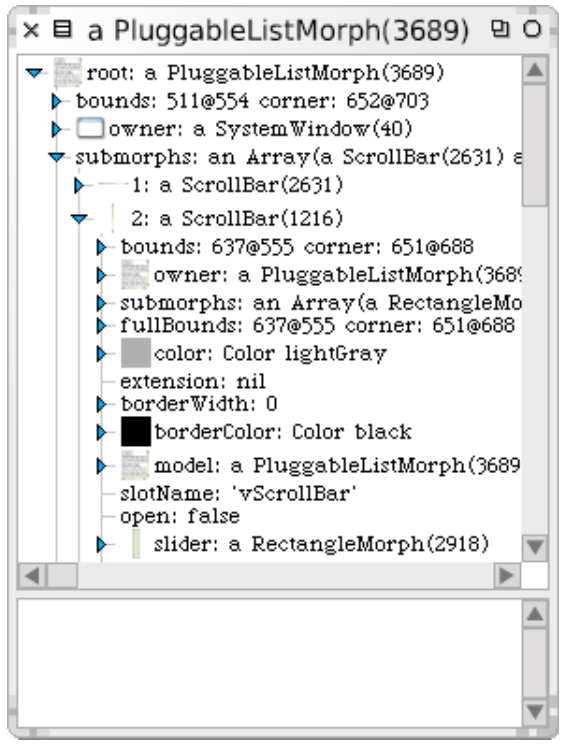


그림 6.23: `PluggableListMorph`를 탐색하기

6.5 디버거 (Debugger)

스퀵의 디버거 는 틀림없이 스퀵도구 모음중에서 가장 강력한 도구입니다. 디버가는 디버깅에만 사용되는 것이 아니라, 새로운 코드를 작성하는 작업에도 사용됩니다. 디버거는 버그를 입력함으로서 시작할 수 있습니다.

 시스템 브라우저를 사용하여, 클래스 `String`에 다음 메서드를 추가하십시오:

Method 6.1: 버그가 있는 메서드

```
suffix
```

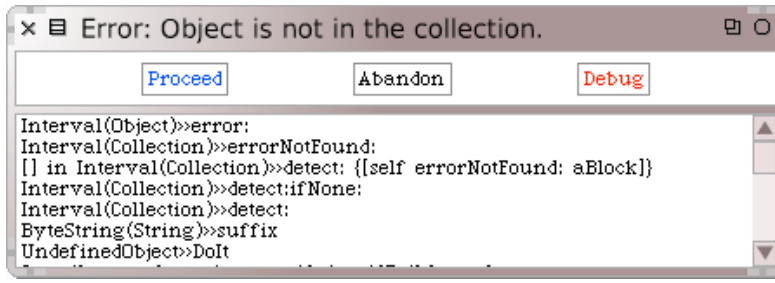


그림 6.24: PreDebugWindow는 버그를 알려줍니다.

```
    "assumes that I'm a file name, and answers my suffix, the
    part after the last dot"
    | dot dotPosition |
    dot := FileDirectory dot.
    dotPosition := (self size to: 1 by: --1) detect: [ :i | (self
    at: i) = dot ].
    ↑ self copyFrom: dotPosition to: self size
```

물론 작업한 메서드는 당연히 작동할 거라 예상하고, SUnit test 를 작성하는 대신에, Workspace에서 `'readme.txt' suffix` 를 입력한 다음 print it (p) 을 진행합니다. 이야! 대단하네요? 기대했던 결과인 `'txt'`를 얻는 대신에, 그림 6.24 에서 보이는 것처럼, PreDebugWindow 를 볼 수 있습니다.

`PreDebugwindow`에는 어떤 오류가 발생했는지에 대해 알려주는 타이틀바(title-bar)가 있으며, 오류를 발생시켰던 메시지의 stack trace 상태를 출력합니다. trace의 출력은 아래쪽부터 시작해서 위쪽으로보면 됩니다. `UndefinedObject>>DoIt` 은 실행되는 컴파일된 코드를 나타내며, 사용자가 Workspace에서 'readme.txt' suffix 를 이용해서 스퀵에게 print it을 요청한 시점에서 실행됩니다. `UndefinedObject>>DoIt` 에서 확인할 수 있는 `'readme.txt'` suffix 라는 코드는 당연히, ByteString 객체(`'readme.txt'`)에게 메시지 suffix 를 전송한다는 의미입니다. 이런 과정들을 거쳐 String 클래스에서 ByteString 으로 상속된 suffix메서드가 실행이 됩니다; `ByteString(String)>>suffix` 에 관련된 모든 정보는 stack trace의 다음줄부터 부호화(encoding) 됩니다. stack이 진행되면서, suffix 매서드는 그 안에서 자신에게 detect:메시지

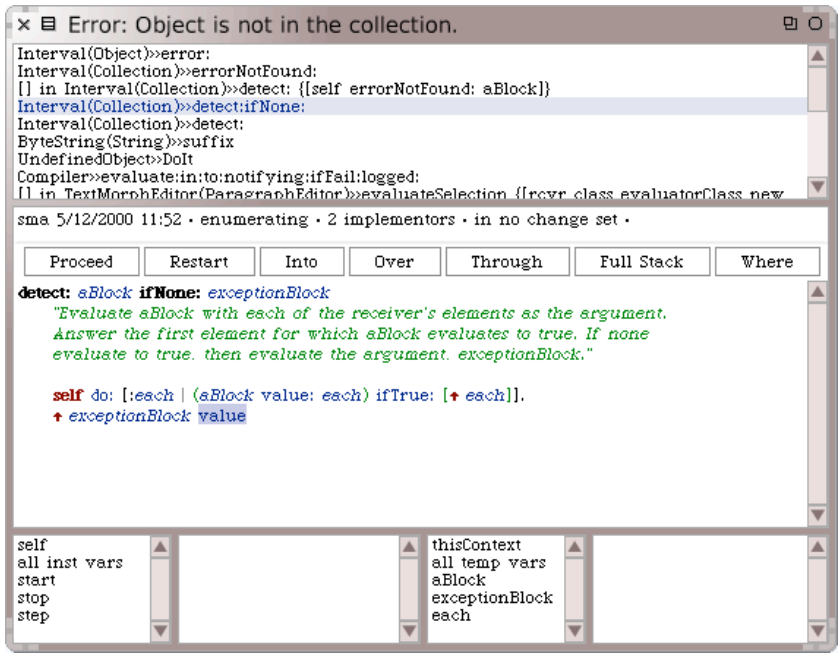


그림 6.25: 디버거

를 보내고 결국 `detect:ifNone`에 의해 errorNotFound가 전송됨을 확인할 수 있습니다.

점(dot)을 발견하지 못한게 문제가 됐습니다. 자세한 이유를 알아보기위해서 디버거가 필요하기때문에 Debug 버튼을 클릭하도록 합니다.

 당신이 *stack trace*의 정보들중 어떤 라인을 클릭해도, 디버거가 열립니다. 디버거는 문제가 생긴 해당 메서드에 이미 포커스된 상태로 열리게 됩니다.

그림 6.25 에 디버거가 보입니다; 이 디버거는 처음에 매우 어렵게 보이지만, 사용하기는 꽤 쉽습니다. 타이틀 바와 상단 패널은 `PreDebugWindow` 에서 보았던 것들과 매우 흡사합니다. 오류가 일어난 실행부분이 당신의 이미지에 그상태 그대로 존재한다는것은 중요한 사실입니다. 대기상태기는 하지만요.

stack trace (추적) 의 각 라인은 실행 stack의 프레임을 나타내는데 그 프레임이란것은 실행을 연속적으로 수행하기 위해 필요한 모든 정보를 포함하고 있습니다. 실행 stack의 프레임은 인스턴스 변수들, 실행 메서드들의 모든 임시 변수들 및 연산에 관련된 모든 객체를 포함합니다.

그림 6.25 를 보면, 상단 패널에서 detect:ifNone: 메서드를 선택했습니다. 선택한 메서드의 내용은 중앙의 패널에 표시됩니다; 파란색으로 강조된 value 메세지는 현재 보이는 메서드에서 value 메세지를 보냈으며 답변을 기다리고 있음을 나타냅니다.

디버거의 아래에 있는 4개의 패널은 실제로 두 개의 mini inspector(Workspace 패널은 빠진상태) 입니다. 왼쪽의 Inspector는 현재 객체에 관련된 내용을 보여주며, 이 경우 현재 객체는 가운데 패널의 내용중 self가 가리키는 객체를 의미합니다. 상단의 목록에서 다른 stack 프레임을 선택하면, self의 아이덴티티는 변경될 수 있으며, self-inspector의 내용도 바뀔 수 있습니다. 좌하단 패널에서 self를 클릭하면 이미 예상했던대로 self의 값은 인터벌 (10 to:1 by -1) 임을 알 수 있습니다. 디버거의 mini inspectors에서는 Workspace 패널이 필요없는데, 왜냐하면 모든 변수들의 범위는 위쪽의 메서드 패널의 내용에 한정되기 때문입니다; 가운데의 메서드 패널에서 입력을 하거나 또는 표현식들을 선택하고 그것들을 실행해보는 작업을 마음대로 할 수 있기 때문이죠. 언제나, 메뉴 또는 CMD–i 를 사용하면 변경사항을 cancel (l) (취소) 할 수 있습니다.

우측 하단의 Inspector는 현재 시점의 임시 변수를 보여줍니다. 그림 6.25 에서, exceptionBlock의 매개변수로 value가 보내졌습니다.

 이 파라미터의 현재 값을 보기 위해, 우측하단의 *context Inspector*에서 *exceptionBlock* 을 클릭 합니다. 클릭하면 *exceptionBlock*의 값이 *[self errorNotFound:]* 임을 알려줄겁니다. 변수의 값으로 해당되는 오류메세지가 보이는건 당연한 일입니다.

그건 그렇고, 디버그창 아래의 mini inspector에 보이는 변수들중 한가지에 대해 full inspector 또는 오브젝트 탐색기를 열기 원하는 경우에는, 변수의 이름을 더블클릭 (inspect) 을 하거나 또는 변수의 이름을 선택하고 노랑 버

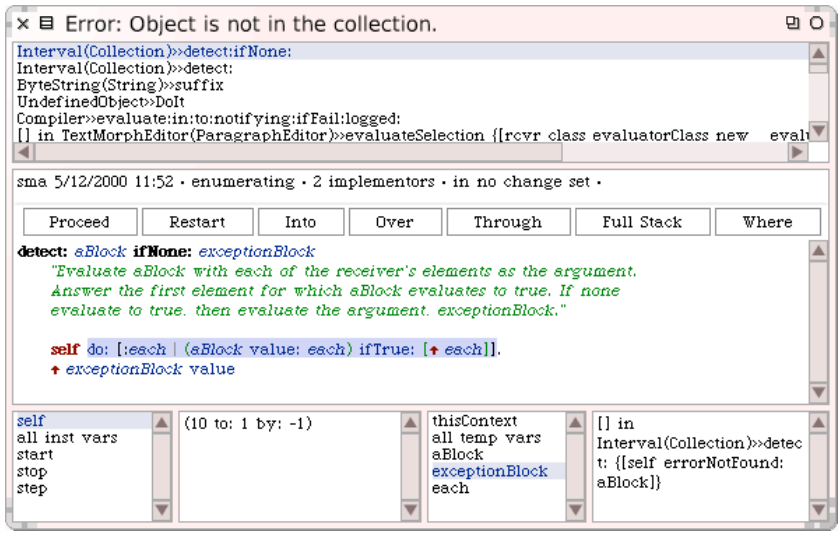


그림 6.26: `detect: ifNone:` 메서드를 다시 시작한 후의 디버거입니다.

튼 메뉴에서 inspect (i) 또는 explore (I) 를 선택하면 됩니다. 이렇게 디버깅중 변수를 inpect 또는 explore 하는 작업은 다른 코드를 실행하는 동안 어떻게 변수가 변하는지 보고싶을때 유용합니다.

위쪽의 stack trace창에서 suffix 부분을 클릭한후 메서드가 표시되는 패널을 보면, 문자열 'readme.txt' 에 대해 메서드의 끝에서 두 번째 라인이 점(dot)을 찾아낼거라 생각했습니다만, 메서드가 동작되는 일련의 작업은 마지막 라인까지 진행되지 못했음을 알 수 있습니다. 스퀵은 디버깅중 역실행을 허용하지 않고, 메서드를 처음부터 재시작하게 하며, 사실 이런방법은 객체를 변형시키는게 아니라 새로운 객체를 만드는 코드에서 매우 잘 동작합니다.

 Restart를 클릭하면, 가운데 패널에서 현재 *stack trace*를 통해 선택한 메서드의 실행이 다시 진행되며 과정이 첫 번째 표현*(statement)*으로 돌아가는걸 확인할 수 있습니다. 파란색으로 강조된 부분을 통해 다음 메시지가 `do:` 에 전송될것이라는걸 보게됩니다 *(그림 6.26 을 보십시오.)*

Into 와 Over를 통해서 두가지 서로다른 실행방법을 진행할 수 있습니다.

만약 Over 버튼을 클릭하면, 스퀵은 오류가 없을 경우, 첫 번째 단계에서 현재 선택된 메시지 전송(이번 경우엔 do:)을 실행합니다. 이후 Over 를 다시 누르면 현재 메서드에 내에서 다음차례의 메시지 전송을 진행하는 위치로 진행되며, 그 대상은 value:가 됩니다. value는 디버깅을 시작한 지점이기때문에 더이상 도움이 되지 않을겁니다. 이제 해야 할 일은 do: 메서드가 사용자가 찾으라고 지정한 문자를 찾지 못하는 이유를 알아내는 것입니다.

 Over를 클릭하고 그림 *6.26* 에 보이는 상황으로 돌아가기 위해 Restart 를 클릭하십시오.

 Into를 클릭하면, 스퀵은 강조된 메시지 전송부분 으로 이동합니다. 이번의 경우는 `Collection>>do:` 가 되겠군요.

이렇게 메서드로 이동을 하는것도, 많은 도움이 되지는 않습니다: 사실 `Collection>>do`가 망가지지 않았다는 것에 대해 어느 정도 자신감이 있으니까요. 버그는 메서드에서 스퀵에게 요청한 작업 내에 있을 확률이 높습니다. Through는 다음같은 경우에 사용하기 유용한 버튼입니다: 이제 do: 메서드 자체의 자세한 내용 을 무시하고 인자블록들의 실행에 집중해봅시다.

 그림 *6.26* 의 환경에서 Through를 클릭합니다. 이후 우측 하단의 컨텍스트 창에 *each*가 생기면 늘 하던 방식대로 *each*를 선택하십시오. *do:* 메서드가 실행되는 *10*에서 부터 시작되며 진행되는 각각의 카운트를 볼 수 있습니다.

each가 7이 될 때, 우리는 ifTrue:block이 실행될 것을 기대할 수 있지만, 그것은 실행되지 않습니다. 무엇이 틀렸는지 보기 위해, 그림 6.27 처럼, each의 값이 7일때 value의 Into 실행으로 이동합니다.

Into를 클릭한 이후, 그림 6.28 처럼 메서드 내에서 강조된 위치를 확인할 수 있습니다. 이 상황은 마치 suffix 메서드로 돌아간 것처럼 보이게되는데, 왜냐하면 현재, detect:에 대한 인수로서 제공된 suffix인 블록을 실행하고 있기 때문입니다. 만약 우측하단의 context inspector에서 i를 선택한다면,

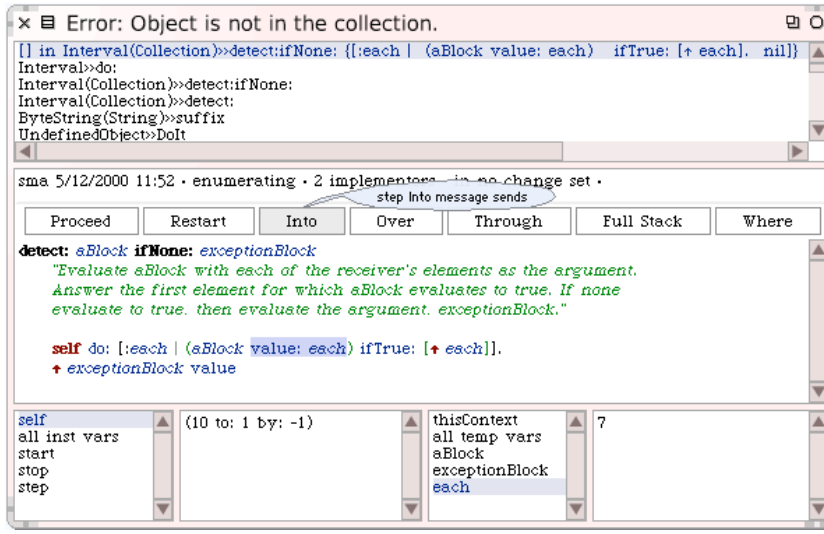


그림 6.27: Through 버튼으로 do: 메서드의 단계를 여러 번 진행한 후의 디버거

선택한 i의 현재 값을 보실 수 있으며, 당신이 만약 지금까지 잘 따라오셨다면, 그 값은 반드시 7이 되어야 합니다. 그 다음, 좌측 하단의 self 변수가 있는 인스펙토에서 self의 해당 구성요소를 선택하면 됩니다. 그림 6.28 에서, 문자열의 구성요소 7이 문자 46임을 볼 수 있으며, 이것은 실제로 dot문자입니다. 만약 context inspector에서 dot를 선택한다면, 변수 dot의 값이 '.' 임을 보게됩니다. 그리고 지금 왜 이것들이 같지 않은지를 확인했습니다: dot은 String인에 비해서 'readme.txt'의 일곱번째 문자는 Character이기 때문입니다.

이제 버그를 확인했고, 수정해야 할 내용은 명확합니다: 'readme.txt' 에 대한 검색을 시작하기 전에 dot를 character 로 변환시켜야만 합니다.

디버거에서 `dot := FileDirectory dot asCharacter` 라고 올바르게 할당하는 내용으로 코드를 변경한후, accept 를 진행해서 변경사항을 적용시킵니다.

현재 detect: 내부의 블록 내부에서 코드를 실행하고 있기 때문에 여러 개의

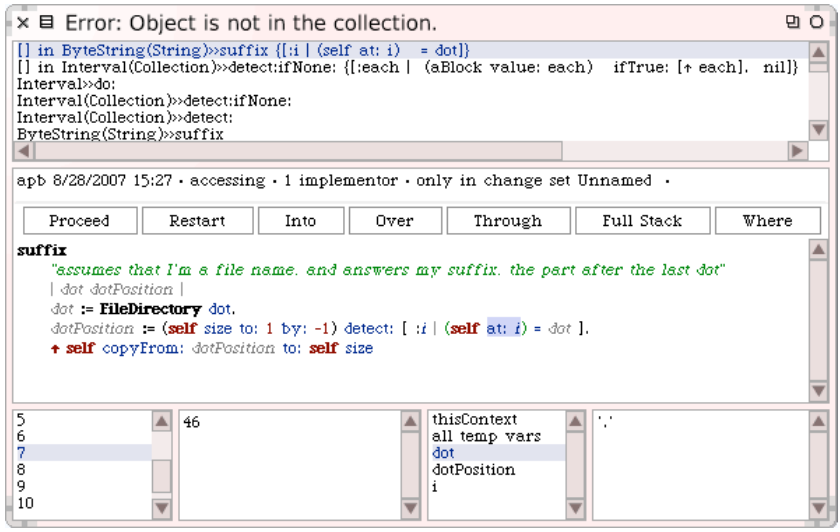


그림 6.28: `'readme.txt' at: 7`가 `dot`와 동일하지 않은 이유를 보여주는 디버거

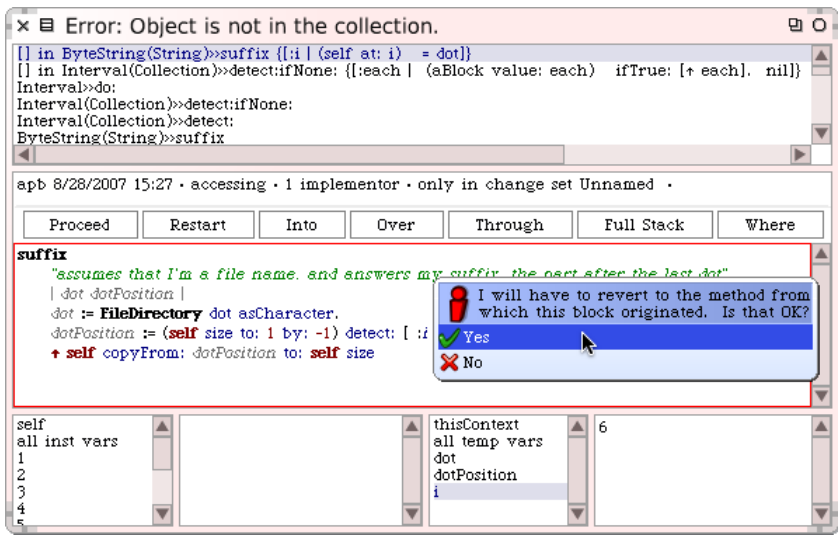


그림 6.29: 디버거에서 `suffix` 메서드를 변경하기 : 내부 블록으로부터 exit에 대한 confirmation을 요청

stack 프레임은 변경사항을 적용하기위해 포기해야만 합니다. 스퀵은 수정작업이 사용자가 원하는 것인지를 물어보며(그림 6.29 를 보십시오) 만약 yes 를 클릭한다고 가정하면, 새로운 메서드를 저장(그리고 컴파일)할 것입니다.

 Restart를 클릭한후 Proceed 버튼을 눌러 진행합니다: 디버거 창이 사라지며, 프로그램식 `'readme.txt' suffix` 의 처리가 완료된후, 원하던 결과물인 `'.txt'`를 `'readme.txt'` *suffiix* 를 실행한 원래의 *Workspace*에 출력합니다.

이 답변은 정확한 것일까요? 안타깝게도 확신 있게 말할 수는 없습니다. Suffix(접미사)는 .txt가 되어야 할까요? 또는 txt가 되어야 할까요? suffix에 있는 메서드 주석은 정확한것이라고 할 수는 없습니다. 이런문제를 어느정도 피하는 방법은 답변을 정의하는 SUnit test를 작성하는 것입니다.

Method 6.2: `suffix` 메서드에 대한 간단한 테스트

```
testSuffixFound
    self assert: 'readme.txt' suffix = 'txt'
```

SUnit 을 사용하는 방법은 Workspace에서 동일한 테스트를 실행하는 것보다 좀더 많은 노력이 필요합니다만, SUnit을 사용하면 실행이 가능한 문서로서 테스트를 저장하고, 다른 사람들이 그 테스트를 실행하기 쉽게 만들어줍니다. 게다가, 만약 StringTest 클래스에 메서드 6.2 를 추가하고, Test Runner에서 SUnit의 테스트세트를 실행한다면, Workspace에서 입력할 때보다 디버깅 오류를 더 빠르게 불러낼 수 있습니다. Sunit은 실패한 검증사항*the failing assertion* 에 대해 디버거를 엽니다만, 해야할일은 아까처럼 복잡하지 않으며 단지 디버거의 stack을 보고 stack 프레임을 한단계 낮추고, 낮춘 프레임의 내용에서 테스트를 Restart 하고, suffix 메서드로 Into 하면, 그림 그림 6.30 에서 볼수있듯이, 오류를 수정할 수 있습니다. 이렇게 오류를 수정한 후, 그 다음으로 해야 할 작업은 단지 Sunit Test Runner에 있는 Run Failures 버튼을 클릭하고, 지금 진행중인 테스트를 확인(confirm)하는 것입니다.

여기에 위의 메서드보다 더 발전된 테스트 내용이 있습니다:

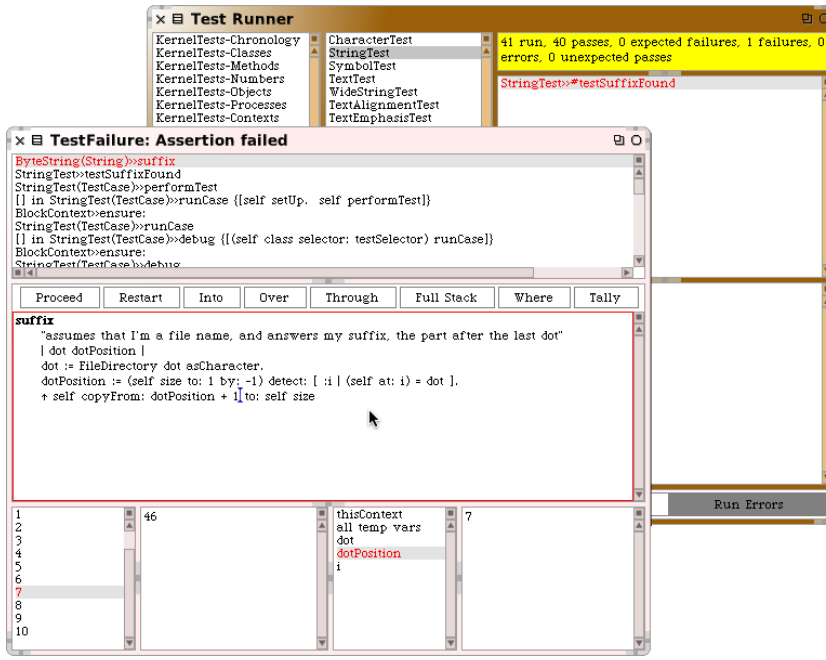


그림 6.30: 디버거에서 `suffix` 메서드 변경하기 : SUnit assertion failure이후, the off-by one 오류를 수정하기.

Method 6.3: `suffix` 메서드를 위한 더 나은 테스트

```
testSuffixFound
    self assert: 'readme.txt' suffix = 'txt'.
    self assert: 'read.me.txt' suffix = 'txt'
```

왜 이 테스트가 더 좋은걸까요? 이 테스트는 대상 문자열에 하나 이상의 dot(점)이 있는 경우, 테스트메서드를 읽는[5] 사람에에게 메서드가 무엇을 해야 할지를 알려주기 때문입니다.

디버거에 오류와 주장 실패 (assertion failures)를 잡아내는 작업 뿐만 아니라, 디버거에 들어가는 몇 가지 다른 방법들이 있습니다. 사용자가 무한 루프

[5]이미 앞부분에서 설명했듯이 Test unit을 작성하는것은 실행가능한 문서로서의 의미이기때문에 이런 표현을 쓰지 않았나 싶습니다

에 들어가는 코드를 실행하면, 그 코드를 인터럽트하고, *'CMD–. (dot)*을 (dot은 마침표 또는 구두점이라고 부르죠)[6]입력하여 연산진행중에 디버거를 열 수 있습니다. 또한 디버거를 열기위해 의심되는 코드에 self halt 를 삽입하는 방법도 있습니다. 예를들어, suffix 메서드를 편집한 내용을 보도록 하죠.

Method 6.4: `suffix` 메서드에 `halt` 삽입하기.

```
suffix
    "assumes that I'm a file name, and answers my suffix, the
    part after the last dot"
    | dot dotPosition |
    dot := FileDirectory dot asCharacter.
    dotPosition := (self size to: 1 by: --1) detect: [ :i | (self
    at: i) = dot ].
    self halt.
    ↑ self copyFrom: dotPosition to: self size
```

우리가 이 메서드를 실행할 때, self halt의 실행은 우리가 진행할 수 있는 장소에 pre-debugger를 불러오거나 또는 디버거로 들어가 변수들을 보고, 연산을 한 단계씩 진행하며 (step), 코드를 수정합니다.

지금까지의 모든 내용은 디버거에는 해당됩니다만, suffix 메서드에 대해서 전부 설명이 된건 아닙니다. suffix 메서드를 실행했을때 문자열에서 아무런 dot(점)이 없는경우에 첫 버그가 발생된다는걸 알아야 하거든요. 이렇게 버그가 일어나는 원하는건 아니기때문에, 이번 예제에서는 dot(점)이 문자열에 없는경우, 어떤 일이 일어나야 할지를 지정하기 위한 두 번째 테스트 메서드를 추가하도록 하겠습니다.

Method 6.5: `suffix` 메서드를 위한 두 번째 테스트: 타켓은 `suffix`를 갖고 있지 않습니다.

```
testSuffixNotFound
    self assert: 'readme' suffix = ''
```

 메서드 *6.5* 를 클래스 `StingTest`에 있는 테스트 세트에 추가하고, 그 테스트를 통해 오류를 발생하는 것을 지켜봅니다. *SUnit*에서 오류가 있는 테

[6] CMD–S HIFT-.(dot)를 입력하여 언제든지 비상용 디버거를 불러올 수 있다는 것을 알아두는것도 중요합니다.

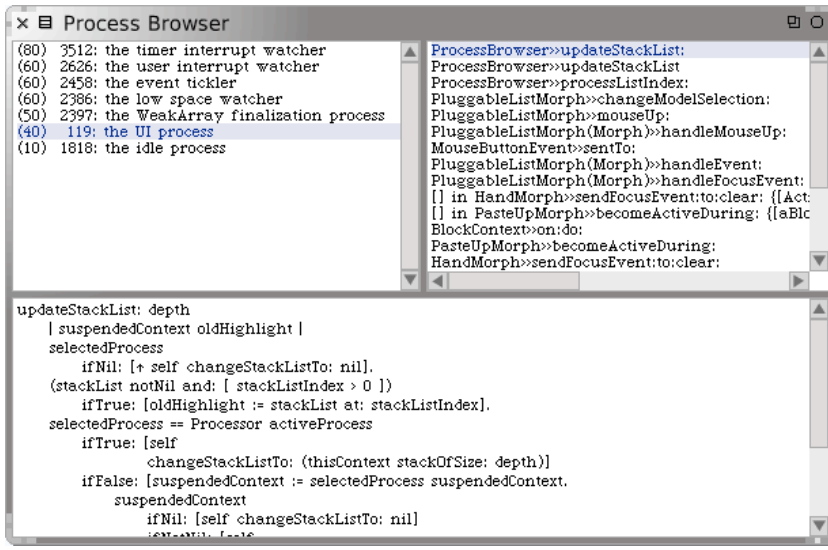


그림 6.31: 프로세스 브라우저(The Process Browser)

스트를 선택해서 디버거로 들어간후, 코드를 수정하여, 테스트를 통과되게끔 합니다. 오류를 수정하는 가장 쉽고 정확한 방법은, *suffix* 메서드의 `detect:` 대신, `detect:ifNone:`을 이용해서 `detect:ifNone:` 두번째 인수를 단순히 문자열의 크기를 반환하는걸로 처리하는 겁니다.[7]

SUnit 에 대해서는 7장 에서 좀 더 배우도록 하겠습니다.

6.6 프로세스 브라우저

스몰토크는 멀티-쓰레드 시스템(multi-threaded system) 입니다: 현재 사용하고있는 이미지에는 동시에 실행되고있는 (쓰레드로 알려진) 많은수의 가벼운 프로세스들이 있습니다. 미래에는 스퀵가상머신이, 물리적 시스템에서 Multi Processor(SMP) 를 사용할 수 있는경우, 이것들을 이용하게 되겠지

[7] 제 경우는 이걸 `detect:[ :i | (self at: i) = dot ] ifNone:[self size]`. 이런식으로 처리했습니다. 일단 에러는 안나는군요. 맞게처리한건지는 잘 모르겠습니다.

만, 현재의 스퀵가상머신에서는 프로세스 동시실행은 시분할 방식으로 구현되어있습니다.

프로세스 브라우저는 스퀵에서 실행중인 다양한 프로세스를 볼 수 있게 해주는 디버거의 사촌격입니다. 그림 6.31 의 스크린 샷을 봐주세요. 상단 왼쪽 패널은 스퀵에서 실행중인 모든 프로세서를 우선순위 80(priority 80)인 time interrupt watcher에서부터 우선순위 10인 idle process 까지 우선순위순서대로 나열합니다. 물론, 당신이 실제로 실행되것을 확인할 수 있는건 UI process 밖에 없으며, 모든건 단일 cpu에서 작동하고 있습니다; UI process 외의 다른 프로세스는 다른 이벤트처리등을 위해 대기하고 있죠. 기본적으로 화면에 보여지는 프로세스들은 정적입니다; 노랑버튼메뉴의 `turn on auto-update (a)` 옵션을 사용하면 프로세스 브라우저의 내용을 자동으로 업데이트 할 수 있습니다.

먄약 왼쪽 상단 패널에서 프로세스를 선택한다면, 선택한 프로세스의 stack trace 정보가 상단 오른쪽 패널에 디버거처럼 디스플레이 됩니다. 만약 오른쪽 상단의 stack 프레중 하나를 선택한다면, 해당되는 메소드가 하단 패널에 표시됩니다. 프로세스 브라우저는 self와 thisContext를 위한 mini-inspectors는 없지만, stack 프레임에서 사용할 수 있는 노랑 버튼 메뉴 아이템들은 디버거의 mini-inspector 기능과 동일한 수준의 기능을 제공합니다.

6.7 메서드 찾기

스퀵에는, 메시지를 검색하는 작업을 도와주는 2개의 도구가 있으며, 두개 도구 모드 플랩에서 드레그하여 꺼낼 수 있습니다. 이 2개의 도구는 인터페이스와 기능이 서로 다릅니다.

1.9절 에서 이미 설명했던, 메서드 파인더[method finder] 는 이름 또는 기능으로 메서드를 찾을 때 사용할 수 있습니다. 하지만 이렇게 메서드를 찾아도, 메서드의 내용을 보려면, 메서드 파인더는 새로운 시스템 브라우저를 열게됩니다. 이런작업은 압도적으로 빠르게 진행됩니다.

메시지 이름 브라우저*message names browser* 는 좀더 제한된 검색 기능을 갖고 있습

그림 6.32: 선택자*selectors* 의 이름중 `random`이라는 문자열이 포함된 모든 메서드들을 보여주는 메서드 이름 브라우저

니다. 사용하려면 검색 상자에서 찾기원하는 메시지 셀렉터의 이름중 일부를 입력해야 하며, 검색을 시작하면 브라우저는 그림 6.32 에서 보이는 것처럼 메서드들의 이름 안에 해당되는 단어를 포함하고 있는 모든 메서드들을 나열합니다. 이렇게 제약된 기능이지만 이 도구는 제대로 된 브라우저입니다: 만약 왼쪽 패널에 있는 검색된 메서드 중 하나를 선택하면 해당되는 이름의 메서드를 가진 클래스-메서드 들이 오른쪽 패널에 나열되며, 하단 패널에서 원하는 클래스-메서드의 내용을 볼 수 있습니다. 클래스 브라우저처럼, 메시지 이름 브라우저의 하단에는 선택한 메서드 또는 해당되는 클래스에 대한 다른 브라우저들을 열기 위해 사용되는 버튼 바가 포함되어 있습니다.

6.8 변경 세트와 변경 정렬자

스퀵에서 여러분이 작업할 때마다, 메서드와 클래스에에 생긴 변경 사항들은 변경 세트(a change set)에 기록됩니다. 이 기록은 새로운 클래스 만들기, 클래스 이름 다시 만들기, 카테고리 변경하기, 현존하는 클래스에 메서드 추가하기 등을 포함한, 모든 중요한 작업들에 대한 내용입니다. 그렇다고해서 아무거나 쓸모없는것까지 포함되는건 아닙니다. 예를들자면, 당신이 워크스페이스에서 새로운 전역변수를 할당한다고 하죠. 변수생성은 변경세트에 추가되지 않습니다.

항상, 많은 변경 세트들이 존재합니다만, 이중 단 한 개만이 현재에 대한 변경 세트 이며, 그것은 이미지에 가해지고 있는 변경 사항들을 수집합니다. 당신은

World ▷ open… ▷ simple change sorter 를 사용하거나 또는 *Tools* 플랩 밖으로 Change Set 아이콘을 드래그해서 어떤 변경 세트가 현재에 대한것인지를 확인할 수 있고, 또한 모든 변경 세트를 검사할 수도 있습니다.

그림 6.33 에서 변경세트 브라우저를 확인하세요. 타이틀 바는 어떤 변경 세트가 현재에 대한 것이며, 브라우저를 열었을 때, 해당 변경 세트가 선택되었다는 것을 보여줍니다.

다른 변경 세트들은 상단 왼쪽 패널에서 선택될 수 있으며, 해당 영역에서의 노랑버튼 메뉴를 이용하면 다른 변경 세트를 현재 상태로 만들 수 있게 하거니 새로운 변경 세트를 만들 수 있게 됩니다. 상단 우측 패널은 선택된 변경 세트(그것들의 카테고리들과 함께)에 의해 영향을 받을 모든 클래스들을 열거합니다. 목록의 클래스들 중 하나를 선택하면, 그 작업은 가운데 패널에 있는 변경 세트(클래스의 모든 메서드들이 아닌)에 있는 클래스의 메서드들의 이름을 출력하며, 중앙의 목록에서 메서드 이름을 선택하면 하단 패널에 메서드 정의를 보여줍니다. 이런정보들은 변경 세트를 나타내는 데 사용되는 객체의 구조에 저장되어 있지만, 클래스의 생성등이 변경세트의 일부인지까지는 브라우저에서 알 수 없다는걸 참고해주세요.

변경세트 브라우저에서는 해당항목의 노란버튼메뉴를 이용해서 변경세트의 내용에서 메서드와 클래스들을 삭제 할 수 있습니다. 하지만, 변경 세트에 대

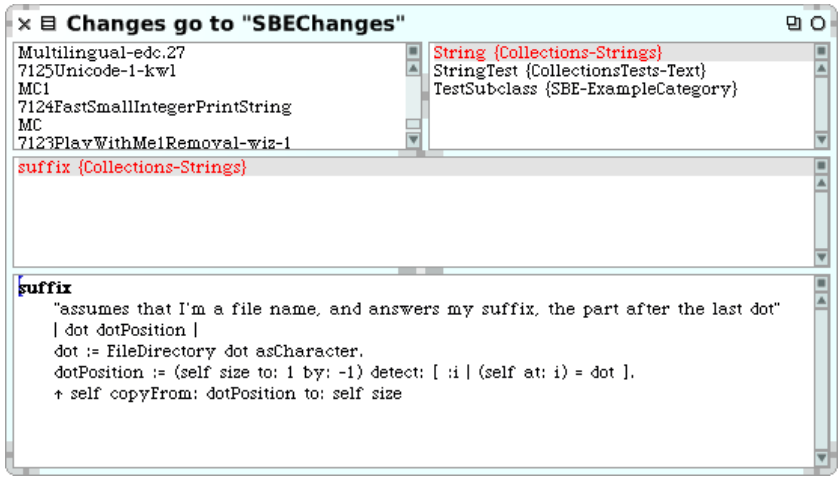


그림 6.33: 변경 세트 브라우저

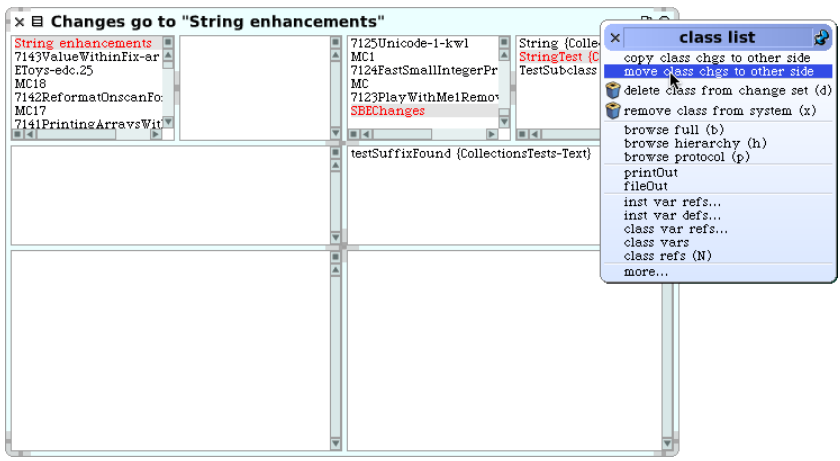


그림 6.34: 변경 정렬자

해 보다 세밀한 편집을 진행하기 위해서는, 그림 6.34 에 보이는 처럼, *Tools* 플랩 또는 메뉴에서 `World ▷ open… ▷ dual change sorter` 선택을 한 후 사용해야만 합니다.

듀얼 변경 정렬자*Dual Change Sorter* 는 기본적으로 두 개의 변경 세트 브라우저를 나란히 정렬 한 모습이며, 양쪽은 각기 다른 변경 세트, 클래스 또는 메서드를 살펴볼 수 있습니다. 이 레이아웃은 그림 6.34 에 보이는 것 처럼 듀얼 변경 정렬자의 주요 기능이며, 현재 스크린샷은 한 개의 변경 세트에서 다른 변경 세트로 변경 사항들을 이동시키거나 복사하는 기능을 진행하는 과정을 보여주고 있습니다.

당신은 왜 변경 세트의 구성에 관해 관심을 가져야만 하는지 의아해 할 수도 있습니다. 이유인즉슨, 변경 세트가 스퀵에서 다른 파일 시스템으로 코드를 내보내고, 내보내진 파일 시스템에서 다른 스퀵이미지 내부로 코드를 보내거나 또 다른 비-스퀵스몰토크로 내보내는 작업에 필요한 단순한 메커니즘을 제공하기 때문이죠. 변경 세트 내보내기는 "filing-out" 으로 알려져 있으며, 모든 변경 세트와 브라우저에서 클래스 또는 메서드 중 하나에서 노랑 버튼 메뉴를 사용하여 사용할 수 있습니다. 파일내보내기를 반복하면 파일의 새로운 버전이 만들어기는 하지만, 변경 세트 브라우저들은 몬티첼로같은 버전을 관리하는 도구가 아닙니다: 변경 세트 브라우저들 종속성을 지속적으로 파악하지는 않거든요.

몬티첼로 가 나타나기 전에는, 변경 세트(change sets)는 스퀵사용자들 사이에서 코드를 교환하는 주요 수단이었습니다. 이전의 스몰토크프로그래머들은 변경세트의 단순성(파일 내보내기로 만들어지는 파일은, 텍스트 에디터로 편집을 권해드리지 않는다해도, 그 자체로 텍스트 파일일 뿐입니다.)과 이동성을 활용하였습니다. 시스템에 직접적인 관련-몬티첼로에서 진행하기에는 준비가 덜된것들-이 없는 많은 차이점들을 변경세트로 만드는건 만드는건 쉽습니다.

변경 세트를 몬티첼로 패키지와 비교하는경우, 변경세트의 중요한 단점은, 변경세트는 종속성의 개념이 지원되지 않는다는 겁니다. 파일로 내보내진 변경 세트는 그 변경 세트를 불러오고 들어갈 모든 이미지를 변경하는 동작들의

모음입니다. 변경 세트를 성공적으로 로드하려면 이미지가 변경세트를 로딩하기 적합한 상태에 있어야 합니다. 예를 들자면, 변경 세트는 아마도 클래스에 메서드를 더하기 위한 동작을 포함하고 있겠죠. 이런 변경 작업은 오직 메서드가 추가될 대상 클래스가 이미지 내부에서 이미 존재하는 경우에만 정상적으로 수행될 수 있습니다. 이와 비슷한 경우로, 변경 세트가 클래스 이름을 다시 지었거나 재 범주화를 시킬수도 있는데, 이런 작업은 분명히 대상이 되는 클래스가 이미지 안에 현존할 때에만 수행될 것이고, 메서드들을 파일 내보내기 하는경우에도, 내보내진 메서드를 로드할때 해당되는 메서드들이 가져온 이미지 안에 존재하지 않는, 없는 클래스에 선언된 인스턴스 변수들만을 사용하게 될겁니다. 문제는 변경 세트들은 파일로 가져온(filed in) 이전 이미지의 조건들을 명백하기 가지고 있는게 아니라는 겁니다: file in과정이 아무문제없이 잘 작동되기를 바라는것 외에는 방법이 없으며, 뭔가 문제가 있을때 대부분 이해하기 힘든 에러메시지와 stack trace등의 결과가 나오게 되겠죠. file in이 잘 진행된다고 해도 현재의 변경세트가 강제로 아무 메시지 없이 조용히 다른 변경세트를 취소해버릴수도 있습니다.

반대로, 몬티첼로 패키지들은 패키지파일 내부에서 서술문 방식으로 코드를 표현합니다: 몬티첼로 패키지들은 내부적으로 이미지의 상태를 로드된 후의 상태가 되도록 표현해야 합니다. 이런 내부의 작업은 몬티첼로로 하여금 사용자에게 충돌에 대한 경고(두 개의 패키지들은 모순되는 최종 상태를 요구합니다)를 진행할 수 있게하며, 종속성 순서에 따라 순서대로 패키지를 로드할 수 있도록 해줍니다.

몬티첼로 패키지에 비해 결점이 있지만, 변경 세트들은 여전히 나름대로 쓰이는곳이 있습니다. 별도로, 살펴보기를 원하거나 아마도 사용하기를 원하는 변경 세트들을 인터넷에서 찾을 수 있습니다. 이런 이유들로 다음에는 변경 정렬자를 사용해서 변경 세트를 file out 하고, 이렇게 만들어진것들을 어떻게 file in(파일에서 가져오기) 하는지 알려드릴겁니다. 파일에서 가져오는 작업에는, 별도의 도구인, 파일 브라우저가 필요합니다.

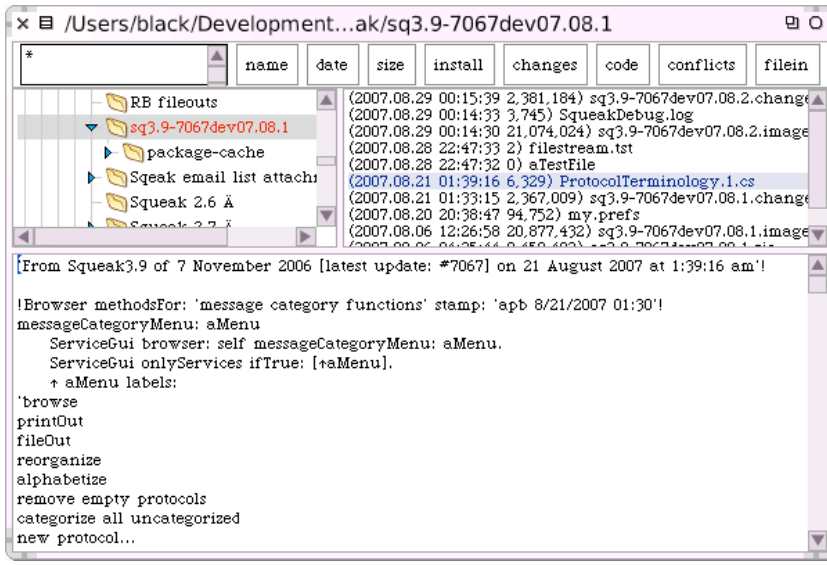


그림 6.35: 파일 목록 브라우저(A file list browser)

6.9 파일 목록 브라우저 (The File List Browser)

파일 목록 브라우저는 사실 스퀵으로부터 파일 시스템[the file system (그리고 FTP 서버들)]을 검색하기 위한 일반 목적 도구(general-purpose tool)입니다. 파일목록 브라우저는 World ▷ open⋯ ▷ file list menu 를 사용하거나 도구 플랩에서 언급된 브라우저를 드레그하여 열 수 있습니다. 물론 파일목록 브라우저를 이용해서 볼 수 있는 것은 로컬 파일 시스템의 컨텐츠들이 기본이 되겠습니다만, 전형적인 모습은 그림 6.35 에 보이는것과 같습니다.

처음 파일 목록 브라우저를 열면, 브라우저는 여러분이 스퀵을 시작한 현재 디렉토리를 대상으로 열립니다. 타이틀 바(the title bar)는 대상이된 디렉토리에 대한 경로를 알려줍니다. 왼쪽에 보이는 더 큰 패널은 오래된 방법이기는 하지만 파일시스템의 경로를 찾는 작업에 사용됩니다. 디렉토리를 선택하면, 디렉토리에 포함된 파일들은(디렉토리가 아닌) 오른쪽에 표시됩니다. 파일 목록은 브라우저창의 왼쪽상단에 있는 작은 입력상자에 유닉스 스타일

패턴 (Unix-style-pattern) 을 입력함으로써, 필터링을 적용할 수 있습니다. 여기서 사용되는 패턴은 첫 글자로, 모든 이름을 매치하는 * 이지만, 얼마든지 패턴을 변경해서 다른 문자열을 타이핑하고 적용할 수 있습니다.(*는 사용자가 입력한 패턴의 앞뒤에 붙여서 사용될 수 있다는걸 기억해두세요.) 파일들의 정렬순서는 name, date 그리고 size 버튼들을 사용하여 변경할 수 있습니다. 버튼의 규칙이 적용된 이후 정렬에 대한 나머지 순서는 브라우저에서 선택된 파일들의 이름을 기준으로 합니다. 그림 6.35 에서 파일이름은 suffix(접미사) .cs를 포함하고 있기때문에, 파일 목록 브라우저는 해당되는 파일이 변경세트 (Change set) 인 것으로 추정하고, 해당파일을 install 하고, 파일에서의 Changes 를 검색하고, 파일의 Code 를 검사하며, 그리고 현재 시스템의 변경세트로 코드를 filein (이 버튼은 파일의 이름에서 파생된 이름을 가진 새로운 변경세트에 해당파일을 file in 작업을 수행합니다) 하기 위한 기능버튼들을 제공합니다. 아마도, Conflict 버튼은 이미지안에 현존하는 코드와 충돌된 변경세트의 변경사항들에 관해 알려줄거라고 예상할 수 있지만, 그렇지는 않습니다. 그 대신, conflict 버튼은 파일이 제대로 로드되지않을 가능성이 있는, 파일에대한 잠재적인 문제 (line feed 기호같은것들) 들만을 점검합니다.

파일의 내용을 참고하는게 아니라, 파일의 이름에 따라 버튼들의 표시가 결정되기때문에, 가끔 원하는 버튼이 나타나지 않는 경우도 있습니다. 그렇지만 전체 옵션 세트는 노랑 버튼을 사용한 more··· 메뉴에 항상 나타나므로, 버튼의 표시에 대한 예비수단으로 사용하면 되겠습니다.

정확한건 아니지만, Code 버튼은 변경세트로 작업을 할 때 가장 유용하게 쓰일거같습니다. 이 버튼은 변경세트 파일 (the change set file) 의 컨텐츠들 위에서 브라우저를 열어줍니다. 예시는 그림 6.36 에 나와 있습니다. 파일 컨텐츠 브라우저는 카테고리들, 클래스들, 프로토콜들과 메소드들을 보여드리지 않는 다는 것을 제외하면 시스템 브라우저와 동일합니다. 각 클래스를 위해, 브라우저는 시스템 상에 클래스가 있는지의 여부와 파일에서 클래스가 정의되었는지의 여부 (그러나 정의들 (definitions) 이 동일한 가의 여부는 제외) 를 알려줍니다. 파일컨텐츠 브라우저는 변경세트 파일의 내부에 대해 (그림 6.36 에 보이는 것 처럼) 각 클래스에 있는 메소드들을 보여드리고, 파일에

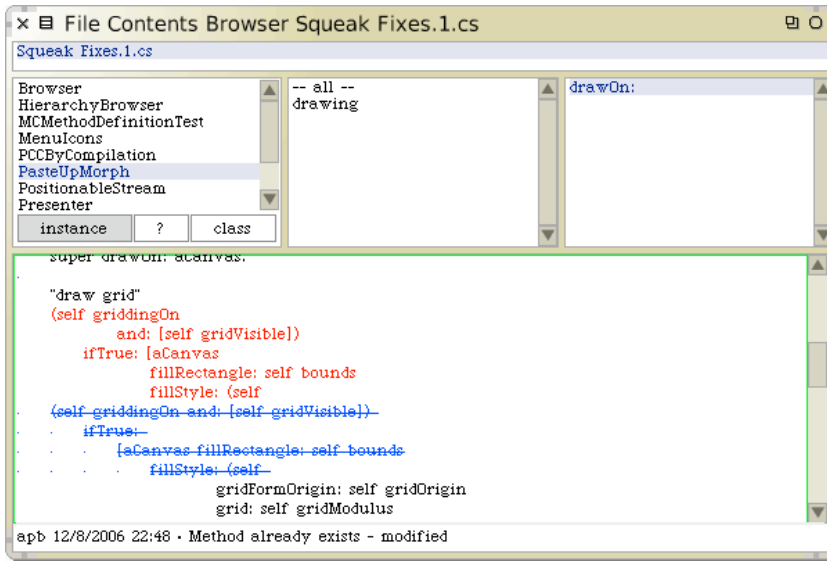


그림 6.36: 파일 컨텐츠 브라우저 (A File Contents Browser)

있는 현재 버전과 버전 사이의 차이점들에 대해 보여줍니다. 상단의 3개의 패널에서 사용할 수 있는 노랑 버튼 메뉴 아이템들은 사용자가 전체 변경 세트 또는 대응 클래스, 프로토콜 또는 메소드를 대상으로 file in을 진행할 수 있도록 해줍니다.

6.10 스몰토크에서, 코드는 잃어버릴 수 없습니다

스퀵에서 충돌은 자주 일어납니다: 스퀵은 실험적인 시스템으로서, 스퀵의 작동에 있어 핵심적인 것들을 포함한 모든 것들에 대한 변경을 허용합니다.

 고의로 스퀵에서 충돌을 보고싶다면, 오브젝트를 *nil*이 되도록 시도하십시오.

하지만 좋은 소식은 강제로 스퀵을 충돌시키고 작업중인 이미지의 마지막 저장된 버전으로 돌아간다 하더라도, 결코 어떤 작업도 잃어버리지 않을거라는

점이며, 몇시간정도 이전으로 돌아가게 될겁니다. 걱정할 필요는 없으며 사용자가 실행한 모든 코드들은 .chages file에 저장되어있기 때문이죠. 이게 전부입니다! 이 파일에는 프로그래밍을 할 때에 클래스에 여러분이 더한 코드뿐만이 아니라 워크스페이스에서 사용자가 실행하는 한줄한줄의 코드까지 포함되어있으니까요.

여기 이전에 작업되었던 코드를 얻는 방법이 있습니다. 사실 코드를 얻어야할 필요가 생기기전에 굳이 이런 방법을 알아야할 필요는 없습니다. 하지만 이 방법이 필요할때 여기서 원하는 내용을 찾을 수 있을겁니다.

가장 안좋은 상황의 경우 .changes 파일에 텍스트에디터를 사용하면 되겠지만, 파일의 용량이 몇메가정도로 크기때문에 속도문제등을 고려해서 그리 권장하지는 않습니다. 스퀵은 에디터를 쓰는것보다 나은 방법을 제공합니다.

당신의 스퀵코드를 다시 얻어내는 방법

스퀵을 가장 최근의 스냅샷으로부터 다시 시작하시고, World ▷ help ▷ useful expressions 을 선택합니다. 이 기능은 유용한 표현식들이 있는 워크스페이스를 열게됩니다. 첫 번째 3개는, 복구를 위해 가장 유용하게 쓸 수 있습니다.

```
% Smalltalk recover: 10000.
ChangeList browseRecentLog.
ChangeList browseRecent: 2000.
```

만약 ChangeListbrowseRecentLog를 실행하면, 검색을 원하는 history의 범위를 결정할 수 있습니다. 보통, 마지막 스냅샷 만큼 멀리 되돌아갈 수 있는 범위로 검색을 하면 충분할 겁니다. (ChangeList browseRecent: 2000 을 편집해서 원하는것과 비슷한 효과를 얻을 수 있으며 2000이라는 숫자는 trial 과 error 를 사용하는것과는 다릅니다.)

최근변경브라우저*recent changes browser* 에서 판단, 제안, 변경등을 마지막스냅샷으로 되돌리면, 그동안 스퀵이미지를 저장하기 전까지 작업한 모든 목록을 얻을 수 있습니다. 노랑 버튼을 사용하면 이렇게 얻어진 목록에서 원하는 아이템들을 지울 수 있습니다. 아이템들을 삭제하다가 만족할만한 상태가 됐을 때, 남아 있는 것들에 대해 file in 을 진행함으로서, 새로운 이미지에 변경사

항들을 결합시킬 수 있습니다. file in 을 하기 전에, 일반적 변경세트 브라우저the ordinary change set browser 를 사용하여 현재의 변경세트에서 새로운 변경세트를 시작하는 것은 좋은 방법이며, 그렇게 함으로써 여태까지 모든 복구된 코드는 새로운 변경세트로 들어갑니다. 그 다음 이렇게 새로만들어진 변경세트를 file out 할 수 있습니다.

최근변경브라우저에서 사용할 수 있는 유용한 기능은 `remove doIts` 입니다. 일반적으로, doIts에 해당되는 작업들은 변경세트에 저장되지 않기때문에 File in(그리고 재실행)을 할 수 없습니다. 하지만 예외가 있습니다. 변수생성과는 다르게 클래스생성은 `doIt` 하는 시점에서 변경세트에서 확인이 가능합니다. 특정 클래스에 대한 메서드를 file in 한다면 그전에 클래스는 존재되어 있어야 합니다. 메서드 file in 과정에서 새로운 클래스를 만들어야 하는 경우, file-in 을 통해 먼저 클래스를 생성하고, `remove doIts` 를 진행하고 메서드를 file in 해야합니다.

저는 복구를 끝냈을 때, 나만의 새로운 변경 세트를 file out 한후, 이미지를 저장하지 않고 스퀵을 종료했다가, 재시작 해서 새로운 변경 세트 파일이 정확하게 다시 복구되도록 하는것을 선호합니다.

6.11 6장 요약

스퀵에서 효과적으로 개발하려면, 스퀵환경에서 사용 가능한 도구들을 배우는 일에 일정시간은 투자하는 것이 중요합니다.

- 표준 시스템 브라우저는 현재 가지고있는 카테고리, 클래스, 메서드프로토콜 과 메서드들을 검색하고, 새로운 것들을 정의하는 작업에 쓰이는 메인 인터페이스 입니다. 시스템 브라우저는 메시지, 메시지의 버전 등의 발신자(senders) 또는 구현자(implementers)에게 직접 건너뛸 수 있는 여러 가지 유용한 버튼들을 제공합니다.
- 여러 종류의 다양한 클래스 브라우저(옴니브라우저(OmniBrowser)와 리팩토링 브라우저(Refactoring Browser))들과, 클래스와 메서드

를 살펴볼 수 있는 여러 가지 특화된 브라우저(계층도 브라우저와 같은)들 이 있습니다.

- 모든 도구에서, 클래스 또는 메서드의 이름을 강조할 수 있으며, 키보드 바로가기 메뉴 CMD–b 를 사용하여 시스템 브라우저로 즉시 건너뛸 수 있습니다.
- 또한 기본 환경에서, 메시지를 SystemNavigation에 보냄으로써 프로그램적으로 스몰토크시스템을 검색할 수 있습니다.
- 몬티첼로는 클래스들과 메서드 들의 패키지를 내보내고(exporting), 들여오고(importing), 버전을관리(versioning)하고 공유하기 위한 도구입니다. 몬티첼로 패키지는 시스템 카테고리, 하위 카테고리 그리고 다른 카테고리와 관련된 메서드들의 프로토콜들로 구성됩니다.
- 인스펙터와 익스플로러는 현재의 이미지에서 라이브 오브젝트를 검색하고 상호작용하는 작업에 유용합니다. 심지어 라이브 오브젝트들의 모픽 할로*morphic halo* 와 디버그 핸들을 불러오기 위해 파랑 클릭을 해서 정밀 검사(inspecting) 하는데 도구들을 사용할 수 있습니다.
- 디버거는 에러가 발생했을 때, 프로그램의 runtime stack 을 정밀 검사 할 수 있게 해주는 것 뿐만 아니라, 소스 코드를 포함하여 어플리케이션의 모든 오브젝트들과 상호작용할 수 있도록 해줍니다. 대부분의 경우, 디버거내의 지속되는 실행중에 소스코드를 편집할 수 있습니다. 디버거는 특별히 SUnit과 제휴하여 test-first 개발을 지원하기 위한 도구로서 효과적으로 사용됩니다.(7장)
- 프로세스 브라우저는 사용자가 자신의 이미지에서 실행중인 프로세스들을 모니터, 쿼리 수행 그리고 상호작용할 수 있도록 합니다.
- 메서드 파인더와 메시지 이름 브라우저는 메서드를 배치하기 위한 두 가지 도구들입니다. 첫 번째 도구는 사용자가 이름은 확실히 모르지만 예상하는 동작을 검색하려 할 때 보다 유용합니다. 두 번째 도구는 적어도 이름을 일부라도 알고 있을 때, 좀더 진보된 검색 인터페이스를 제공합니다.

- 변경 세트는 현재 사용하고있는 이미지의 소스코드에 대한 모든 변경 사항들의 로그들을 자동으로 기록합니다. 이 변경 세트는 소스 코드의 버전들을 저장하고 변경하는 수단으로서 몬티첼로에 의해 대부분의 상황에서 대체되고 있습니다. 그러나 비록 드물게 발생할 수 있지만, 재난에 가까운 오류들을 복구하는 작업에서는 특별히 유용합니다.
- 파일 목록 브라우저는 파일 시스템을 검색하기 위한 도구입니다. 또한 이 브라우저는 파일 시스템으로부터 소스 코드상에 파일로 가져오기 (`file in`)를 할 수 있도록 해줍니다.
- 사용자 자신의 소스코드를 저장하기 전 또는 몬티첼로로 그 소스 코드를 백업하기 전에 이미지가 충돌할 경우, 변경 목록 브라우저를 사용하여 가장 최근의 변경사항들을 언제든지 복구해낼 수 있습니다. 복구 후 이미지를 다시 실행해서, 이미지의 가장 최근의 사본으로 정리하는 변경 사항을 선택하면 됩니다.

제 7 장

SUnit

7.1 개요

SUnit 은 테스트의 생성과 배치를 지원하는 최소규모의 강력한 프레임 워크입니다. 그 이름에서 추측할 수 있듯이 SUnit 의 디자인은 단위 테스트에 집중되어 있지만, 사실상 통합 테스트와 기능 테스트를 위해 사용될 수 있습니다. SUnit 은 본래 Kent Beck이 개발하였으며, Joseph Pelrine이 확장하였고 또 다른 사람들이 7.6절 에 우리가 설명드릴 내용인 자원의 개념을 통합하였습니다.

테스트 실행과 테스트 주도 개발에 대한 관심은 스퀵또는 스몰토크에 제한된 것은 아닙니다. 자동화 테스트 실행은 애자일 소프트웨어 개발 운동의 전형적인 특징을 이루었으며, 모든 소프트웨어 개발자들의 소프트웨어의 질 향상에 대한 관심은 그것을 채택하도록 만들 것입니다. 실제로 많은 언어를 사용하는 개발자들은 Uniet 테스트 능력의 진가를 알아보기 시작하였으며, xUnit 의 버전은 Java, Python, Perl, .NET, Oracle 과 오라클을 포함한 많은 다른 언어들로 존재합니다. 이 장은 SUnit 3.3(이 글을 쓸 당시 최신버전)에 대해 기술하였으며, 업데이트를 발견할 수 있는 SUnit 의 공식 웹은 sunit.sourceforge.net입니다.

테스트 또는 테스트 수트의 구축 모두 새로운 것은 아니며, 모든 사람들은 테스트가 에러를 잡아내는 좋은 방법임을 알고 있습니다. 테스트를 핵심 실행법으로 만들고 자동화된 테스트를 강조한 eXtream 프로그래밍은, 테스트 실행을 프로그래머들이 싫어하는 정기적인 업무가 아닌, 생산적이고 재미있게 만드는 작업이 되도록 도움을 주어 왔습니다. 스몰토크커뮤니티는 오랜 테스트 실행의 전통을 갖고 있으며, 그 이유는 개발의 증강 형식은 그 프로그래밍 환경에 의해 지원되기 때문입니다. 전통적인 스몰토크개발환경에 있는 프로그래머들은 테스트들을 메서드가 마무리 된 후 곧바로 워크스페이스에서 작성할 것입니다. 때때로 시험상으로 실행해 볼 메서드의 머리 부분에서 주석으로 통합할 수 테스트들과 또는 약간의 설정이 필요한 테스트들은 클래스에서 예제 메서드로 포함될 수도 있습니다. 이 실행방식의 문제점들은 워크스페이스에서의 테스트들은 코드를 수정하는 다른 프로그래머들이 사용할 수 없다는 것이며, 주석들과 견본 메서드들은 이러한 점에서 더 낫지만, 그것들을 추적하고, 자동으로 시작하는 것은 여전히 쉽지 않습니다. 실행하지 않은 테스트는 버그를 찾는 작업에 있어 사용자를 도울 수 없습니다. 더욱이, 예제 메서드는 기대한 결과의 리더(reader)를 알려주지 않으므로, 예제를 실행할 수 있으며 아마도 놀랄만한 결과를 보실 수 있을 것이지만, 관찰된 동작이 정확한지에 관하여서는 알 수 없을 것입니다.

SUnit은 유용한데, 자체 점검에 대한 테스트 작성을 우리에게 허용해주기 때문입니다: 테스트는 스스로 올바른 결과로 무엇이 되어야 하는지를 정의합니다. 또한 SUnit은 테스트들을 그룹들로 조직하고, 테스트들이 반드시 실행햐야할 컨텍스트를 기술하며, 테스트들의 그룹을 자동으로 실행하는 작업을 도와줍니다. 2분 남짓한 시간이면, SUnit을 사용하여 테스트들을 작성할 수 있기때문에, 워크스페이스에서 테스트를 위한 작은 코드 조각을 작성하는 대신에, SUnit을 사용하여 모든 저장된 내용들과 자동으로 실행 가능한 테스트들을 만들어 활용하실 것을 권장합니다.

이 장에서 우리는, 왜 우리가 테스트를 시행하고 무엇이 좋은 테스트를 만드는지에 대해 토론부터 시작하겠습니다. 그 다음 SUnit을 사용하는 방법을 알려주는 작은 예시들을 시리즈로 제공할 것입니다. 마지막으로, 같이 SUnit의 실행을 살펴보다보면, 스몰토크에서 도구가 스스로를 지원하는 힘을 어떻게

사용하는 지를 이해할 수 있을겁니다.

7.2 테스트 수행이 중요한 이유

불행하게도, 많은 개발자들이 테스트는 시간낭비라고 생각합니다. 무엇보다도, 이런 개발자들은 버그를 만들지 않으며, 오직 다른 프로그래머들이 버그를 만든다고 믿습니다. 우리들중 대부분은 "내가 좀 더 많은 시간이 있으면 테스트들을 작성할 꺼야"라고 자주 말하기도 합니다. 만약 당신이 결코 버그를 만들지 않고, 그렇게 작성된 코드가 앞으로도 결코 변경되지 않을 것이라고 한다면, 실제로 테스트는 시간낭비일 수 있십니다. 하지만 정말 그렇다면, 대부분의 경우에 있어 만들어진 어플리케이션은 하찮거나, 제작자 및 그 누구도 사용하지 않는다는 의미도 됩니다. 미래를 위한 투자로서 테스트를 생각해 보시기 바랍니다: 테스트들의 suite 를 가진다는건 현 시점에서도 유용하지만, 그보다 앞으로 만들게될 어플리케이션 또는 어플리케이션 환경에서, 어플리케이션을 실행하고 변경할 때 훨씬 더 유용하게 될겁니다.

테스트들은 여러가지 역할을 수행합니다. 첫번째로, 테스트는 테스트가 담당하는 기능에 대한 문서를 제공하게 됩니다. 게다가, 이렇게 제공되는 문서는 활동적[1]입니다: 테스트가 제대로 통과된다면 테스트로 구현되는 문서화가 최신상태라는 의미가 됩니다. 두번째로 테스트들은 개발자가 패키지에 대한 변경사항으로 인해 시스템에 있는것들이 정상임을 확인하는것을 도와주며, 정상적일것이라는 예상에 대한 좋지않은 결과가 나오면, 개발자들이 문제가 있는 부분을 찾는것을 지원하게 됩니다. 마지막으로, 프로그래밍과 동시에 테스트를 작성하는-또는 그 이전에 테스트를 작성하는등-것은 테스트를 실행하는데 목적을 두는것이 아니라, 프로그래밍시 설계시 원하는 기능, 그리고 구현보다 고객에서 보여질지에 대해 더 많이 생각하게 합니다. 테스트를 먼저-코드를 만들기전에-작성함으로써 실행할 기능, 클라이언트 코드와 상호작용하는 방식 그리고 예상되는 결과에 컨텍스트를 기술하는것이 필요해집니다. 이렇게 해서 코드는 개선될 것입니다: 시도해 보십시오.

[1] 실행이 가능한 문서라는 점. 6.5에서 관련부분이 조금 설명되어있습니다

현실적으로 봤을때 어플리케이션의 모든측면을 테스트할 수는 없습니다. 하나의 어플리케이션 전체를 모두 테스트하는 것은 단순히 불가능한 작업이며, 테스트의 목적이 될 수 없습니다. 훌륭한 테스트 suite를 갖고 있어도, 어플리케이션에 버그는 생길 수 있으며, 이렇게 생긴 버그는 시스템을 손상시킬 기회를 엿보며 잠복하고 있을 수 있습니다. 만약 이러한 현상을 발견하시면, 그 버그를 활용하세요! 버그를 발견하자마자, 버그발생내용대로 테스트를 작성해서 버그를 드러나게 하고, 테스트를 실행해서 오류가 생기는 부분을 지켜보십시오. 이제 버그 수정을 시작하면 됩니다: 테스트를 진행하다보면 언제 테스트 작업을 끝내야 할지를 자연스럽게 알 수 있을겁니다.

7.3 좋은테스트를 만들려면 어떻게 해야할까요?

좋은 테스트 작성은 연습으로 가장 쉽게 학습될 수 있는 기술입니다. 최대의 이익을 얻게 해줄 테스트의 속성을 함께 살펴봅시다.

1. 테스트는 반드시 반복실행이 가능해야 합니다. 테스트는 원하는 만큼 반복이 가능해야 하며 항상 동일한 답을 얻을 수 있어야만 합니다.

2. 테스트는 사람의 개입 없이 실행되어야만 합니다. 밤에도 테스트를 실행할 수 있어야만 합니다[2].

3. 테스트는 기능에 이어질 수 있는 내용[3]이어야 합니다. 그리고 각 테스트의 코드는 할당된 테스트를 모두 담당할 수 있어야 합니다. 테스트는 시나리오로서 작동해야 하며, 당신 또는 다른 사람이 해당되는 테스트의 기능을 파악하기위해 테스트코드를 읽을 수 있는 상태가 되어야 합니다.

4. 테스트는 반드시 테스트들이 대상으로 하는 기능보다는 적은 회수로 수정되어야 합니다: 당신의 어플리케이션을 수정할 때마다 관련된 모든 테스트를 바꾸는것은 번거로운일이죠. 이렇게 적은 회수로 테스트

[2]밤에 자동으로 돌린다고해도 수행이 가능한정도로 자동화가 되어야한다는 의미입니다.
[3]아마도 원문의 story가. 스토리보드정도의 의미가 아닐까 합니다

를 수정하기 위한 한가지 방법은 테스트를 만들때 테스트할 클래스의 공용 인터페이스에 기반을 두고 만드는겁니다. 만약 테스트가 필요할 만큼 메소드가 충분히 복잡하다고 느껴지는 경우, 개별적 "helper"를 위한 테스트를 작성하는 것은 괜찮지만, 좀더 나은 실행법을 생각해 내었을 때, 그러한 테스트들은 바뀌거나 완전히 버려질 수도 있다는 것을 반드시 생각해 두어야 할것입니다.

위의 (3) 번 항목을 참고로 해서 테스트의 목적을 좁혀, 테스트해야 할 기능과 테스트의 수가 비례하도록 합니다: 시스템의 일부를 변경해도 모든 테스트가 이상이 생기는게 아니라 일부의 테스트만 실패하도록 되어야 합니다. 이런진행방법은 상당히 중요합니다. 왜냐하면 100 개의 테스트 실패는 10개의 테스트 실패보다 심각한 내용이기 때문입니다. 그렇지만, 전체테스트에 모두 실패하는 상황이 불가능한것만은 아닙니다: 특별히, 무언가를 변경한뒤 오브젝트의 초기화나 테스트의 설정에 문제가 생기는경우, 관련된 모든 테스트는 제대로 작동하지 않습니다.

eXtreme Programming 에서는 코드를 작성하기 전에 테스트를 먼저 작성할 것을 주장하고 있습니다. 이런주장은 소프트웨어 개발자로서의 깊은 본능과 맞지 않을수도 있습니다. 프로그래머들은 대부분 이렇게 말할겁니다: 계속하고 시도해! 우리는, 코드가 무엇을 코딩하기 원하는지 돕고, 작업을 끝났을 때를 알도록 돕고 그리고 클래스의 기능을 개념화하고 그 클래스의 인터페이스를 디자인하는 작업을 돕기 전에, 테스트를 먼저 작성해야 한다는 것을 발견하게 되었습니다. 더 나아가 테스트를 먼저 만드는 개발방법은 코드를 만드는 작업이 빨리 진행되도록 독려합니다. 왜냐하면 테스트로 인해 중요한걸[4] 잊어버리는것에 대해 두려워하지 않아도 되기때문입니다.

[4] 테스트가 특정 기능을 만드는목적을 잊지 않게 해준다는 의미

7.4 예제로 보는 SUnit

SUnit 의 세부사항을 살펴보기 전에, 차례차례 예제를 살펴 보도록 하겠습니다. 여기서는 클래스 세트를 테스트하는 예제를 사용하게 될것입니다. 아래의 내용대로 코드 입력을 시도해 보십시오.

1 단계 : 테스트 클래스를 만듭니다.

 먼저 여러분은 `ExampleSetTest`라 불리는 `TestCase`의 새로운 하위 클래스를 만들어야 합니다. 두 개의 인스턴스 변수를 추가하면 여러분의 새로운 클래스가 다음처럼 보이게 됩니다 :

Class 7.1: 견본 세트 테스트 클래스

```
TestCase subclass: #ExampleSetTest
    instanceVariableNames: 'full empty'
    classInstanceVariableNames: ''
    poolDictionaries: ''
    category: 'MyTest'
```

`ExampleSetTest` 클래스에 Set 클래스의 테스트를 정의합니다. 이 클래스에는 테스트를 진행할 컨텍스트를 정의합니다. 여기서 컨텍스트는 full set 과 empty set에 해당되는 full 과 empty 라는 두개의 인스턴스 변수로 이루어집니다

클래스의 이름이 절대적으로 중요한것은 아닙니다만, 관례를 따라, Test로 끝내는것을 권장합니다. 만약 `Pattern`으로 지칭되는 클래스를 정의하는 경우 대응되는 테스트 클래스는 `PatternTest` 라는 이름이 되며, 두 개의 클래스들은 브라우저에 함께 알파벳순으로 정리됩니다. (동일한 카테고리에 있다고 가정한다면) 만들어진 테스트 클래스가 `TestCase`의 하위클래스가 되는것은 대단히 중요한 부분입니다.

2단계 : 테스트 컨텍스트를 (test context) 초기화 하기

`setUp` 메서드는 테스트가 실행될 컨텍스트 (the context) 를 정의하며, 초기화 메서드와 (initialize method) 약간 비슷합니다. setUp 메서드는 테스트

클래스(the test class)에서 정의된 각 테스트 메서드(test method)의 실행 전에 먼저 적용됩니다.

 다음과 같이 `empty` 메소드를 정의하고, 인스턴스 변수인 `empty` 에 빈상태의 *Set* 를 대입한후 *full*에 두 개의 요소를 가지는 *Set* 를 대입합니다.

Method 7.2: *fixture* 셋업하기

```
ExampleSetTest>>setUp
    empty := Set new.
    full := Set with: 5 with: 6
```

테스팅에 대한 전문용어로 컨텍스트는 `fixture for the test`[5] 라고 부릅니다.

3 단계 : 몇 가지 테스트 메서드를 작성하기

클래스 ExampleSetTest에서 몇 가지 메서드를 정의해서 몇 개의 테스트를 만들어 보겠습니다. 각 메서드는 한 개의 테스트를 나타내며, 메서드의 이름은 반드시 문자열 'test' 로 시작해야하고, 그렇게 이름을 만들게되면 SUnit이 테스트 suite 로 해당되는 이름들을 수집합니다. 테스트 메서드들은 어떤 인자들도 필요로하지 않습니다.

 다음의 테스트 메서드를 정의하십시오.

testIncludes 라는 이름을 가진 첫 번째 테스트는 Set의 includes: 메서드를 테스트합니다. 테스트는 5를 인자로 하는 include: 5 라는 메세지를 보내면 true가 반환되는것으로 처리하고 있습니다. 이 테스트는 setUp 메서드가 이미 실행(instance의 생성)되었다라는것을 분명한 전제로 하고 있습니다.

Method 7.3: *set* 멤버십 테스트

```
ExampleSetTest>>testIncludes
    self assert: (full includes: 5).
    self assert: (full includes: 6)
```

[5] 텍스트를 고정한다. 또는 고정부.. 정도 되겠습니다만.. 알맞는 표현이 없어서 원문을 그대로 썼습니다

두번째 테스트인 testOccurrences에서는, 이미 5가 포함되어있는 Set(full)에 5 를 더한다고 해도 결과가 그대로 1 인것을 검증합니다.

Method 7.4: 메서드 *7.4:* 상황발생 테스트

```
ExampleSetTest>>testOccurrences
    self assert: (empty occurrencesOf: 0) = 0.
    self assert: (full occurrencesOf: 5)  = 1.
    full add: 5.
    self assert: (full occurrencesOf: 5)  = 1
```

마지막으로 Set(full)에서 5를 제거한다음 5가 더이상 포함되지 않는 Set를 테스트하게됩니다.

Method 7.5: *Testing removal*

```
ExampleSetTest>>testRemove
    full remove: 5.
    self assert: (full includes: 6).
    self deny: (full includes: 5)
```

true가 되어서는 안될 어떤 것을 assertion 할 때, 메서드 deny:가 사용됨을 주의해주시기 바랍니다. aTest deny: anExpression은 aTest assert: anExpression not 과 같은 의미지만, 훨씬 더 알아보기 쉽습니다.

4 단계 : 테스트를 실행합니다

테스트들을 실행하는 가장 쉬운 방법은 SUnit Test Runner 를 사용하는 것이며, 이 도구는 `World ▷ open···` 메뉴를 사용하거나 도구 플랩 (the Tools flap)에서 `TestRunner`를 드레그해서 열 수 있습니다. 그림 7.1 에 보이는 TestRunner는 테스트들의 그룹을 골라 쉽게 실행할 수 있도록 설계되었습니다. 테스트 클래스들 (예를 들어, TestCase의 하위클래스들)를 포함하고 있는 모든 시스템 카테고리들을 나열하며, 이 카테코리들 중 원하는것을 선택하면, 선택된 카테고리에서 포함하고 있는 테스트 클래스들이 그 패널의 오른쪽 패널에 나타납니다. 추상 클래스들은 이텔릭체로 인쇄되어 있으며, 테스트 클래스 계층도 (the test class hierarchy)는 들여쓰기되어 나타나므로, ClassTestCase의 하위클래스들은 TestCase의 하위클래스들보다 더 들여쓰기 되어 표시됩니다.

*Test Runner*를 열고, 카테고리 `MyTest`를 선택한 다음, 선택된 버튼 `Run Selected`를 클릭하십시오.

그리고 다음의 코드를 대상으로 `print it` 을 실행해서 만들어진 테스트를 실행할 수 있습니다: `(ExampleSetTest selector: #testRemove) run.` 이 표현식은 더 짧은 ExampleSetTest run: `testRemove`와 같은 의미입니다. 그리고 일반적으로 메서드 7.6 처럼 브라우저에서 `do it` 으로 테스트 메서드들을 실행해볼 수 있는, 실행가능한 주석을 테스트 메서드들 속에 포함시키죠.

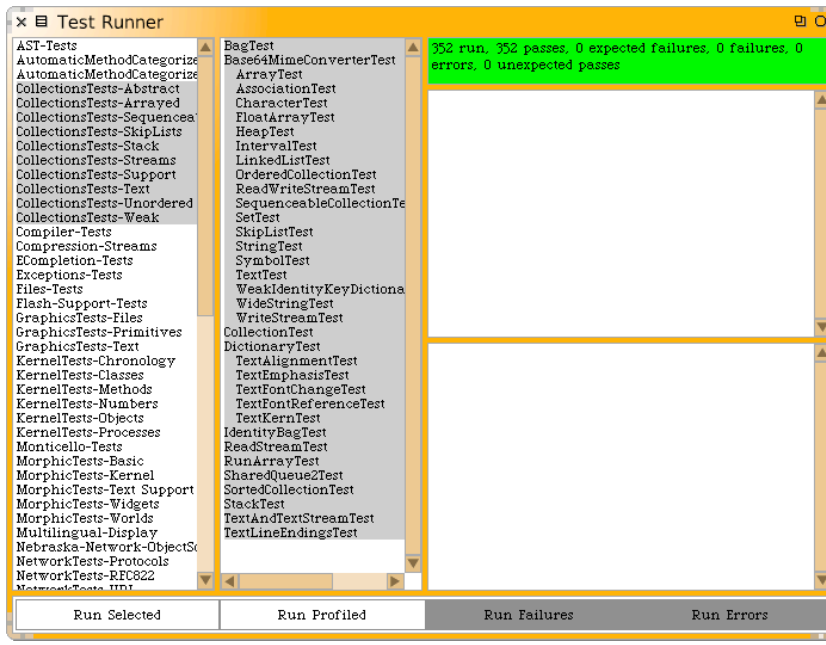


그림 7.1: 스퀵SUnit Test Runner

Method 7.6: 테스트 메서드에서 실행가능한 주석

```
ExampleSetTest>>testRemove
    "self run: #testRemove"
    full remove: 5.
    self assert: (full includes: 6).
    self deny: (full includes: 5)
```

 `ExampleSetTest>>testRemove`에서 버그를 새로 만들고, 테스트를 다시 실행해보세요 예를 들어, *5*를 *4*로 변경해 보시기 바랍니다.

검사과정에서 실패된 테스트는(실패한 테스트가 존재한다면) Test Runner의 오른쪽에 표시되며, 만약 그 테스트가 실패했는지 이유를 보기 위해 디버그를 수행하시려면, 단지 테스트 이름을 클릭하기만 하면 됩니다. 다른 방법으로 다음의 표현식중 하나를 실행해도 됩니다:

```
(ExampleSetTest selector: #testRemove) debug
```

또다른 방법으로는 아래와 같은 방식도 가능합니다.

```
ExampleSetTest debug: #testRemove
```

5단계: 결과의 해석

TestCase 클래스에서 정의된 assert: 메서드는 일반적으로 표현식이 테스트된후 반환값으로 `Boolean` 인자 값을 처리합니다. 반환값이 True 면, 테스트는 통과처리되며 인자가 False 라면 테스트는 실패로 처리됩니다.

하지만 사실 테스트의 결과는 3가지로 나뉠 수 있습니다. 프로그래머가 희망하는 결과는 테스트의 모든 assertions 들이 True가 되는 것이고, 그럴 경우, 테스트들은 통과됩니다. Test Runner에서, 모든 테스트들이 통과될 때, 상단의 막대는 녹색으로 변합니다. 하지만, 테스트를 실행할 때, 좋지않은 결과가 나올 수 있는 경우는 2가지가 있습니다. 가장 확실한 경우는, assertions 중 하나가 테스트를 실패로 만들게되는 False가 될 수 있다는거죠. 다른 경우로는 message not understood 오류 또는 index out of bounds error 처럼,

테스트가 실행되는 동안 일어날 수 있는 몇가지 종류의 오류가 있습니다. 만약 오류가 발생되는 시점에서 테스트 메서드에 있는 assertion 들이 다 실행되지 않는 경우도 있기때문에, 모든 테스트가 실패되었다고 할 수는 없습니다만, 그렇다 하더라도 어떤 테스트에서 문제가 일어났는지는 분명히 확인할 수 있습니다. Test Runner에서 실패한 테스트가 있는경우 상단의 막대는 노랑색으로 바뀌며, 오른쪽의 패널에 나열됩니다. 반면에 테스트 자체에 오류가 있는 테스트들은 막대를 빨강으로 바꾸며, 오른쪽 아래의 패널에 나열됩니다.

 오류와 실패가 일어난 테스트들을 수정하십시오.

7.5 SUnit 활용하기

이 섹션에서는 SUnit 을 사용하는 방법에 대한 좀 더 세부적인 사항들을 다루게 됩니다. 만약 당신이 JUnit[6]과 같은 다른 테스트 프레임워크를 사용해 본적이 있다면, 이후의 내용중 많은 부분이 익숙하실텐데, 왜냐하면 이 모든 프레임 워크들은 SUnit 를 기반으로 시작되었기 때문이죠. 일반적으로 테스트들을 실행하기 위해서 SUnit 의 GUI 를 사용합니다만, GUI 를 쓰고싶지 않을때도 있습니다.

Other assertions

테스트를 만즐때에는 assert: 와 deny: 뿐만아니라, assertions 을 만들기 위해 사용될 수 있는 여러개의 다른 메서드들도 있습니다.

첫번째로 살펴볼 메서드는 `assert:description:` 와 `deny:description:` 으로서 실패의 원인을 알기 힘든경우, 메서드의 두번째 인수로 테스트가 실패한 이유를 알려주는 string 을 지정합니다. 뒤에 살펴볼 7.7절 에 이 메서드에 대한 설명이 있습니다.

그 다음으로는, SUnit 에서 제공되는 테스트진행시 적합한 예외가 발생하는지를 테스트하는 메서드인 `should:raise:` 와 `shouldnt:raise:` 가 있습니

[6] http://junit.org

다. 예를 들어, `self should: aBlock raise:anException`를 실행했을 경우, aBlock의 실행중에 예외 anException가 발생하면 테스트는 성공입니다. 메서드 7.7 에서 should:raise의 사용법을 확인할 수 있습니다.

 이 테스트를 실행해보세요

should:와 shouldnt:의 첫 번째 인수는 처리할 표현식을 포함하고 있는 블록이 되어야 하는것에 주의합니다.

Method 7.7: 오류 발생에 대한 테스트

```
ExampleSetTest>>testIllegal
    self should: [empty at: 5] raise: Error.
    self should: [empty at: 5 put: #zork] raise: Error
```

SUnit 은 다른곳에서도 사용이 가능합니다: SUnit 은 스몰토크의 모든 파생 언어들에서 사용될 수 있습니다. SUnit의 개발자들은 SUnit이 다른곳에서도 사용될 수 있도록 하기위해 여러가지-파생언어에 비의존적일 수 있는 각도에서 고려했습니다. `TestResult class>>error` 클래스 메서드는 구현언어-독립적인 방식을 위해 시스템의 오류 클래스를 해결책으로 사용합니다. 다음과같은 방법을 사용하면 좋습니다: 만약 테스트를 작성할때 어떤 스몰-토크방언에서도 메서드 7.8 처럼 작성한다면 말이죠.

Method 7.8: 간편하게 오류를 다룰 수 있는 방법

```
ExampleSetTest>>testIllegal
    self should: [empty at: 5] raise: TestResult error.
    self should: [empty at: 5 put: #zork] raise: TestResult error
```

 위의 테스트 작성방법을 시도해보시기 바랍니다.

단일 테스트 실행하기

일반적으로, Test Runner 를 해서 작성한 테스트들을 실행할 것입니다. 만약 open… 메뉴 또는 도구 플랩에서 Test Runner를 실행하는게 귀찮다면, Workspace 등에서 TestRunner 를 print it 하면 실행할 수 있습니다.

다음과같이 단일테스트를 실행할 수도 있습니다.

```
ExampleSetTest run: #testRemove  ⟶  1 run, 1 passed, 0 failed,
    0 errors
```

테스트 클래스에서 모든 테스트들을 실행하기

TestCase의 모든 하위클래스들은 suite 라는 메시지에 응답하며, 이 응답은 문자열 "test" 로 시작하는 이름을 가진 클래스에 있는 모든 메서드들을 포함하는 테스트 suite 를 만들게 됩니다. 이렇게 만들어진 suite의 테스트들을 실행하려면 실행을 원하는 suite에 run이라는 메세지를 발송합니다. 예를 들자면 아래와 같습니다:

```
ExampleSetTest suite run  ⟶  5 run, 5 passed, 0 failed, 0 errors
```

반드시 TestCase 클래스의 하위클래스가 되어야 하나요?

JUnit 에서는, test* 메서드들을 포함하고 있는 임의의 클래스로부터, TestSuite를 구축해 낼 수 있습니다. 스몰토크에서도, 이런식의 동일한 작업을 할 수는 있습니다만, 직접 suite 를 만들어야만 하며, 이렇게 만들어진 클래스는 assert:와 같은 모든 필수적인 TestCase 메서드들을 실행하게 될 것입니다. 이런작업은 그리 권장하고 싶지 않네요. 현재 존재하는 프레임웍을 사용하는 것이 더 좋은 방법입니다.(아래에는 직접 suite를 만든경우 테스트를 진행하는 방법에 대한 예입니다)

7.6 SUnit 프레임웍

SUnit은 4개의 메인클레스로 구성되어 있습니다: TestCase, TestSuite, TestResult, TestResource 인데 그림 7.2 에서 확인할 수 있습니다. 테스트 리소스의 개념은 구성하는데 많은 비용이 들기는 하지만 테스트의 시리즈에 리소스로 사용되기 위해 SUnite 3.1부터 등장했습니다. TestResource는 테스트들의 suite이전에 한번만 실행되는 setUp 메서드를 명시합니다; TestResource는 각 테스트 전에 실행되는 `TestCase>>setUp`에 대해서 고유합니다.

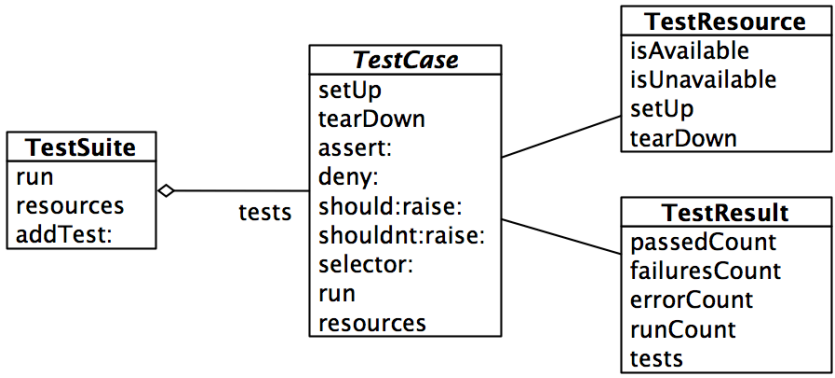


그림 7.2: SUnit 의 핵심이 되는 4개의 클래스

TestCase

TestCase는 하위 클래스가 되도록 설계된 임의의 클래스 이며 그 하위 클래스들 각각은 공통된 컨텍스트(테스트 suite)를 공유하는 테스트들의 그룹을 나타냅니다. 각 테스트는 TestCase, 실행중인 setup, 실행중인 테스트 메서드 자체, 그리고 실행중인 `tearDown` 의 하위 클래스의 새로운 인스턴스를 만들어서 실행됩니다.

컨텍스트는 하위 클래스의 인스턴스 변수들과 메서드 setUp의 특화로 지정되며, 이러한 컨텍스트의 지정 동작은 인스턴스 변수들을 초기화합니다. TestCase의 하위 클래스들은 tearDown 메서드를 재지정할 수 있으며, 이러한 재지정작업은 각 테스트의 실행 후에 호출되고, setUp 을 실행하는 동안 모든 할당된 객체들을 릴리즈 하기 위해 사용될 수 있습니다.

TestSuite

TestSuite 클래스의 인스턴스들은 test cases의 컬렉션을 포함합니다. TestSuite의 인스턴스는 테스트들과 다른 테스트 suite 를 포함하고 있습니다. 즉 테스트 suite는 TestCase와 TestSuite의 하위인스턴스를 포함하고 있다는 게 됩니다. 각각의 TestCases와 TestSuites는 같은 프로토콜을 알아들을 수 있기때문에, 같은방식으로 취급하는것도 가능합니다; 예를 들어, 양쪽다 실

행이 가능하다는 것이죠. 사실 이런것들이 가능한 이유는, TestSuite는 TestCases의 일부와 합성을 통한 composite 패턴의 어플리케이션이기 때문입니다. — composite 패턴에 대한 좀 더 많은 정보를 보시려면 Design Patterns [5]를 봐주시기 바랍니다.

TestResult

TestResult 클래스는 TestSuite 실행의 결과를 나타냅니다. 이 클래스는 통과한 테스트들의 개수, 실패한 테스트들의 개수 그리고 발생한 오류의 갯수 등을 기록합니다.

TestResource

테스트 suite의 중요한 특징 중 하나는 테스트는 각각 독립적이어야 한다는 점입니다: 한 개의 테스트의 실패때문에 다른 테스트들이 대량으로 실패되는 현상이 일어나서는 안되며, 또한 테스트들은 서로의 실행 순서에 영향을 미쳐서도 안됩니다. 각 테스트를 진행하기전에 setUp 을 사용하거나 테스트뒤에 tearDown 을 쓰는건 각 테스트사이의 독립성을 강화하는데 도움이 됩니다.

하지만, 가끔 테스트를 실행하기전 한번 유용하게 사용되어야 하는데 필요한 컨텍스트를 설정하는 작업이 너무나 시간을 잡아먹어서 사용하기 힘든때가 있죠. 더군다나, 이런 테스트 케이스가 테스트에서 사용하는 리소스에 지장을 주지않는다는것을 알게된다면 각각의 테스트를 위해 컨텍스트를 다시 설정하는 작업은 시간낭비입니다; 이런경우 테스트의 각 suite는 한번만 설정하는것으로 충분합니다. 예컨데, 테스트들의 suite가 데이터 베이스를 조회할 필요가 있거나 몇몇 컴파일한 코드에 대한 분석을 시행해야 할 필요가 있다고 가정해 보도록 합니다. 이런경우, 실행할 모든 테스트를 시작하기 전에 데이터 베이스를 먼저 설정하고, 데이터베이스에 대한 연결을 열거나 또는 몇몇 소스 코드를 컴파일하는 작업이 좋은 실행법입니다.

대체 어디로 이러한 자원을 캐쉬해야, 테스트 suite 들이 공유할 수 있을까요? 특정 인스턴스 변수는 TestCase의 하위인스턴스가 될 수 없는데, 왜냐하면 이런 인스턴스(변수)는 오직 테스트를 진행하는 과정에서만 존재되기

때문입니다. 글로벌 변수라면 항상 존재하기때문에 리소스의 용도로 사용이 가능합니다만, 많이 쓰는경우 namespace가 복잡해지며, 글로벌 변수와 테스트사이의 의존성의 강제되는것도, 별도로 명시되는것도 아니기때문에 클로벌변수의 리소스 사용은 권장할만한 방법은 아닙니다. 보다 좋은 방법이 있는데, 필요한 리소스를 몇몇 클래스들의 singleton 오브젝트에 집어 넣는 거죠. 이런 경우에 사용하기 위해 리소스클래스로 TestResource 클래스가 준비되어있습니다. TestResource의 하위 클래스로 생성된 singletone 인스턴스에 대해 current 메시지를 사용할 수 있으며, current 메시지 사용시 해당되는 singleont 인스턴스가 응답하게 됩니다. 반드시 테스트의 리소스가 생성 되고 소멸되도록 하기 위해 setUp 메서드와 tearDown 메서드는 하위 클래스 내부에서 재정의되어야 합니다.

한가지더 : SUnit은 어떤 테스트 suite와 어떤 리소스가 연관되었는지에 관해 어느 정도는 알고 있어야만 합니다. 리소스는 클래스 메서드 리소스를 재정의함으로써 TestCase의 특정 하위 클래스와 연관됩니다. TestSuite의 리소스는 기본적으로 그것이 포함하고 있는 TestCases의 리소스와 묶여집니다.

예제를 보도록 하겠습니다. TestResource의 하위 클래스로서 MyTestResource 를 정의하고, MyTestCase의 클래스 메소드인 resources 를 오버라이드 해서 미리 생성한 리소스인 MyTestResource의 배열이 반환되도록 합니다.

Class 7.9: *TestResource* 하위 클래스의 예

```
TestResource subclass: #MyTestResource
    instanceVariableNames: ''

MyTestCase class>>resources
    "associate the resource with this class of test cases"
    ↑{ MyTestResource }
```

7.7 SUnit의 고급 기능

현재버전의 SUnit에는 TestResource 뿐만 아니라 assertion description string 과 log 지원, 그리고 테스트를 실패한경우 다시시작*resumable* 이 가능한

기능등이 포함되어 있습니다.

Assertion description string

TestCase의 assertion 프로토콜은 프로그래머가 assertion에 대한 명세*description*를 제공할 수 있도록 해주는 몇가지 메소드를 포함하고 있습니다. Description은 문자열입니다; 만약 테스트 케이스가 실패하면 이 Description 문자열은 test runner에 표시됩니다. 물론 이 Description 문자열을 동적으로 만들 수 도 있죠.

```
| e |
e := 42.
self assert: e = 23
    description: 'expected 23, got ', e printString
```

TestCase에서 description 과 관련된 메소드들은 다음과 같습니다:

`TestCase`의 관련 메서드는 다음과 같습니다.

```
#assert:description:
#deny:description:
#should:description:
#shouldnt:description:
```

Log 지원

앞에서 설명한 description 문자열은 Trascript 또는 file stream 과 같은 Stream에 의해 log로 출력될 수 있습니다. 만들어지는 테스트 클래스에 `TestCase>>isLogging` 메서드를 재정의 함으로써 log의 진행여부를 선택할 수 있으며, log 를 진행하는경우 적합한 stream 을 반환하기위해 `TestCase>>failureLog` 메서드를 재정의 함으로써 log의 결과를 어디로 출력할지 선택해야 합니다.

Continuing after a failure(테스트실패 이후의 지속)

SUnit에서는 테스트 실패 이후에, 진행되던 테스트를 지속해야할지에 대한 여부를 결정할 수 있습니다. 이것은 매우 강력한 기능으로써, 스몰토크에 의

해 제공된 예외 메커니즘을 사용게 됩니다. 예제를 들어 이것이 어떤 용도에 사용될 수 있는지를 보겠습니다. 다음 테스트 표현식을 살펴보겠습니다:

```
aCollection do: [ :each | self assert: each even]
```

이번 경우, 진행과정에서 짝수가 아닌 경우 컬렉션의 첫 번째 구성요소를 찾자마자 테스트는 정지됩니다. 그렇다 하더라도, 일반적으로는 테스트가 멈추기를 바라지도 않을거고, 얼마나 많은 구성요소들과 어떤 구성요소들이 짝수가 아닌지 확인을 원할수도 있으며, 아마도 이런 정보들을 지속적으로 log 할 필요가 있겠죠. 이렇게 테스트작업이 끝까지 계속되기 원하는경우, 이 작업을 다음처럼 변경하면 됩니다:

```
aCollection do:
    [:each |
    self
        assert: each even
        description: each printString , ' is not even'
        resumable: true]
```

위의 프로그램식은 실패하는 각 구성요소를 위해 지정된 log 시스템에 메시지를 출력합니다. 그리고 프로그램식은 실패 결과를 쌓아놓지는 않는데, 예를 들어, 만약 assertion이 여러분의 테스트 메서드에서 10번을 실패한다고 해도, 오직 한번의 실패 결과만 확인할 수 있다는 의미가 됩니다. 지금까지 살펴본 모든 assertion 메소드들이 문제가 있는경우 계속해서 실행되는것은 아닙니다; `assert: p description: s` 은 `assert: p description: s resumable: false` 와 같은 의미가 됩니다.

7.8 SUnit의 내부구현

SUnit의 내부구현은 스몰토크프레임웍에 대한 흥미로운 사례가 되기도 합니다. 이제부터 테스트의 실행과정을 통해서 SUnit 내부구현중 핵심이 되는 부분을 살펴보겠습니다.

테스트 실행하기

`(aTestClass selector: aSymbol) run`을 진행해서 프로그램식에 대한 테스트를 해보도록 하겠습니다.

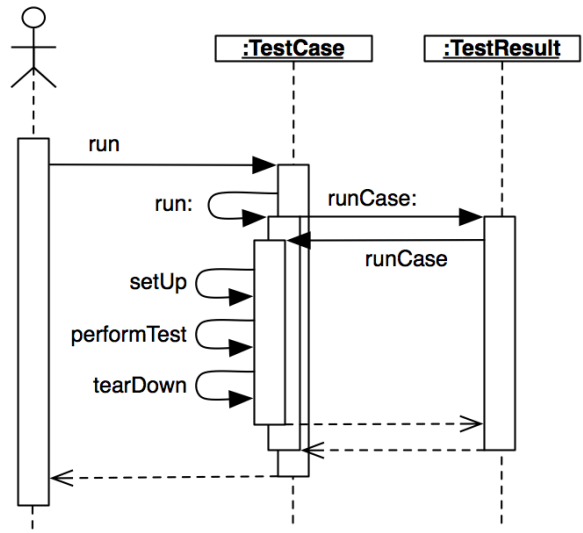


그림 7.3: 테스트 실행하기

`TestCase>>run:` 메서드의 구현을 보면, 테스트의 결과를 모으기 위해서 TestResult의 인스턴스를 만들어서 자기 자신에게 run: 메세지를 전송합니다. (그림 7.3 을 참고해주세요)

Method 7.10: *test case* 실행

```
TestCase>>run
    | result |
    result := TestResult new.
    self run: result.
    ↑result
```

`TestCase>>run:` 메서드는 runCase: 라는 메시지를 테스트 결과로 전송합니다.

Method 7.11: *test case* 를 테스트 결과로 전달하기

```
TestCase>>run: aResult
    aResult runCase: self
```

TestResult>>runCase 메서드는, 인수로서 건네받은 TestCase에 runCase 메세지를 송신합니다. 그 다음 테스트로 에러가 발생한 회수와 실패한 회수, 성공한 회수를 카운트합니다. 예외 핸들러를 설정하고, 예외의 발생과 assertion의 실패에 대비합니다.

Method 7.12: *test case* 오류와 실패를 잡아내기

```
TestResult>>runCase: aTestCase
    | testCasePassed |
    testCasePassed := true.
    [[aTestCase runCase]
            on: self class failure
            do:
                [:signal |
                failures add: aTestCase.
                testCasePassed := false.
                signal return: false]]
                    on: self class error
                    do:
                        [:signal |
                        errors add: aTestCase.
                        testCasePassed := false.
                        signal return: false].
    testCasePassed ifTrue: [passed add: aTestCase]
```

TestCase>>runCase 메서드는 setUp 메시지와 tearDown 메시지를 아래의 예처럼 전송합니다.

Method 7.13: *Test case* 템플릿 메서드 *(Test Case Template Method)*

```
TestCase>>runCase
    self setUp.
    [self performTest] ensure: [self tearDown]
```

TestSuite 실행하기

여러개의 테스트를 한번에 실행하기 위해, 관련된 테스트들을 포함하고 있는 TestSuite에 메시지를 발송합니다. TestCase 클래스는 그것의 메서드들로

부터, 테스트 슈트(test suite)를 구축하기 위해 몇 가지 기능을 제공합니다. 프로그램식 MyTestCase buildSuiteFromSelectors는 MyTestCase 클래스에서 정의된 모든 테스트들을 포함하고 있는 suite 를 리턴합니다. 이 진행 과정에서 핵심은 아래부분입니다.

Method 7.14: 자동 구축 테스트 슈트*(Auto-building the test suite)*

```
TestCase class>>testSelectors
    ↑self selectors asSortedCollection asOrderedCollection
    select: [:each |
        ('test*' match: each) and: [each numArgs isZero]]
```

`TestSuite>>run` 메서드는 TestResult의 인스턴스를 만들고 모든 리소스가 사용가능한지를 확인한 뒤, 그 다음 자신에게 run: 메시지를 전송하게 되는데, 이 작업은 suite에 속한 모든 테스트를 실행시키게 되빈다. 그 이후 모든 리소스가 해제됩니다.

Method 7.15: 테스트 *suite* 실행

```
TestSuite>>run
    | result |
    result := TestResult new.
    self areAllResourcesAvailable
        ifFalse: [↑TestResult signalErrorWith:
                    'Resource could not be initialized'].
    [self run: result] ensure: [self resources do:
                                    [:each | each reset]].
    ↑result
```

Method 7.16: *TestResult* 를 *TestSuite* 로 전달하기

```
TestSuite>>run: aResult
    self tests do:
            [:each |
            self sunitChanged: each.
            each run: aResult]
```

TestResource 클래스와 그것의 하위클래스들은 current 클래스 메서드를 사용하여 접근할 수 있으며, 만들어진 하위 클래스들의 (클래스당 하나의) 인스턴스들을 지속적으로 파악합니다. 이 인스턴스는 테스트가 실행을 마치거나 리소스가 리셋될 때 자동으로 정리됩니다.

리소스 유효성 점검은 클래스 메서드인 `TestResource class>>isAvailable`에서 확인할 수 있듯이 필요할 경우 다시 만들어지는 작업을 가능하게 합니다. `TestResouce`의 인스턴스를 생성 할때, 인스턴스는 초기화되고 메서드 `setUp` 이 호출됩니다.

Method 7.17: 테스트 리소스 유효성 *(Test resource availability)*

```
TestResource class>>isAvailable
    ↑self current notNil
```

Method 7.18: 테스트 리소스 생성

```
TestResource class>>current
    current isNil ifTrue: [current := self new].
    ↑current
```

Method 7.19: 테스트 리소스 초기화

```
TestResource>>initialize
    self setUp
```

7.9 테스트에 관한 몇 가지 조언

테스트의 작동원리는 어렵지 않습니다만, 정작 좋은 테스트를 작성하는 작업은 어렵습니다. 이 아래에 테스트를 설계 하는 방법에 대한 몇 가지 조언이 있습니다.

Unit 테스트를 위한 Feathers의 규칙. 애자일 과정 컨설턴트이며 저자인 Michael Feathers는 다음과 같은 주장을 했습니다.[7]

> 만약 다음에 해당되는경우 작성된 테스트는 Unit 테스트라고 할 수 없다:
>
> - 데이터베이스와 통신하거나

[7] See http://www.artima.com/weblogs/viewpost.jsp?thread=126923. 9 September 2005

- 네트웍을 이용해서 데이터를 주고받거나
- 파일시스템에 접근하거나
- 다른 유닛테스트와 같이 실행할수 없거나 또는
- 테스트를 진행하기 위해 (config 등의 파일을 편집하는등) 현재의 환경에 특별한 작업들을 수행해야 할 경우

이런 작업들이 포함된 테스트가 항상 나쁜 것은 아닙니다. 이따금씩 이런 테스트들을 만들 필요도 있으며, Unit 테스트와 연동되는 형식으로 작성될 수 있습니다. 그렇지만 정말 중요한건 진짜 Unit 테스트와 분리해서 변경이 필요한경우 빠른 테스트셋을 유지할 수 있도록 하는거죠.

위의 작업들이 포함된 Unit 테스트 suite는 시간을 많이 필요로 할 수 있기때문에 이런상황은 만들지 않는것이 좋습니다.

Unit Tests 대 Acceptance Tests. Unit 테스트는 기능의 일부분을 대상으로 하며, 대상이 되는 기능에 있는 버그를 쉽게 파악할 수 있도록 해줍니다. 가능하면 실패할 가능성이 많은 메서드에 대한 Unit 테스트를 만드는걸 진행해보시고, 클래스마다 테스트들에 대한 범주화를 시도해보세요. 그렇지만, 분명한 것은 깊이 반복되거나 또는 복합적인 설정 상황들의 경우에는, 응용프로그램의 시나리오를 따라가는 테스트를 작성하는게 더 쉽습니다; 이러한 테스트들을 허용 테스트(accepttance test) 또는 기능 테스트 라고 부르죠. Feather의 규칙에 해당되지 않는 테스트들은 분명 좋은테스트라고 할 수 있습니다. 그리고 테스트를 시행하는 기능에 따라 허용 테스트들을 범주화 시키십시오. 예를 들어, 만약 컴파일러를 만들려고 하는경우, 대상이되는 개별 소스 언어 정책을 참고해서 제작된 코드에 대한 assertions 을 진행하는 허용 테스트를 만들 수도 있습니다. 이런경우를 대상으로 만들어지는 테스트들은 많은 클래스들을 테스트하게되며, 실행시간도 오래걸리게 되죠. 왜냐하면 이런 테스트들은 파일 시스템을 건드리기 때문입니다. SUnit 을 사용하여 허용 테스트들을 작성할 수 있지만, 작은 변경 사항이 생길때마다 시간이 오래

걸리는 테스트들을 매회 실행되는것은 좋지않기때문에, 이렇게 시간이 오래 걸리는 테스트는 반드시 진짜 unit 테스트들과 구별되어야 합니다.

테스트진행에 대한 Black의 규칙. 시스템상의 모든 테스트들을 봤을때, 몇 가지 속성을 테스트함으로서 결과를 신뢰할 수 있게 만든다는것을 알아야 합니다. 중요한 속성은 빼놓지 말고 테스트되어야 한다는것도 분명한 사실입니다. Black의 규칙은 유용한 속성에 대한 신뢰도를 높이며, 가치를 가지지 못하는 테스트는 시스템에 없어야 한다는 사실을 말합니다. 예를들자면, 같은 속성의 여러가지 테스트들은 쓸모없습니다. 사실 이런 테스트들은 대단히 해롭기까지 하죠 : 이런 테스트들은 테스트들을 읽었을때 클래스의 동작을 추측하는것을 어렵게 합니다. 그리고 코드에 버그가 있는경우 한번에 많은 테스트들에 문제가 생기기 때문에 안좋기도 합니다. 테스트를 작성할 때는 이런 속성들을 주의 깊게 살펴봐야 합니다.

7.10 7장 요약

7장 에서는 왜 코드의 미래를 봤을때 테스트가 중요한 투자인가를 설명하였습니다. 그리고 단계적으로 클래스 세트의 몇몇 테스트들을 정의하는 방법을 설명해드렸습니다. 그 다음, TestCase, TestResult, TestSuite, TestResources 클래스들을 알려드림으로서 SUnit 프레임워크 코어에 대한 개요를 간략히 보여드렸습니다. 마지막으로, 테스트와 테스트 suite의 실행을 따라감으로써 SUnit의 구현방법을 좀 더 자세히 살펴보았습니다.

- unit 테스트들이 최대한 쓸모있는 존재가 되기위해서는 반드시 실행속도가 빨라야 하고, 반복가능해야 하며, 모든 직접적인 인간의 상호작용에서 독립적이어야 하고, 단일 unit 기능성까지 지원할 수 있어야 합니다.
- MyClass 라 지칭되는 클래스를 위한 테스트들은 MyClassTest 로 분류된 클래스에 속하며, TestCase의 하위 클래스가 되어야 합니다.
- setUp 메소드를 사용해서 테스트 데이터를 초기화해야 합니다.

- 각 테스트 메소드는 반드시 “test” 라는 단어로 시작되어야 합니다.
- assertions 를 만들기 위해 TestCase의 메소드인 `assert:`, `deny:` 그리고 다른 것들을 사용하십시오.
- SUnit test runner 도구(툴 바에 있음)를 사용하여 테스트를 실행하십시오.

제 8 장

기본 클래스

스몰토크의 대부분의 마술은 언어자체에 있는 것이 아니라 클래스 라이브러리에 있습니다. 스몰토크로 효과적인 프로그래밍을 하기 위해서는, 클래스 라이브러리가 언어와 환경을 어떤 식으로 지원하는지를 학습해야 합니다. 클래스 라이브러리는 대부분 스몰토크로 작성되었으며, 원하는 클래스가 정의되어있지 않아도 패키지에 새로은 기능의 클래스를 쉽게 추가할 수 있기때문에 확장이 쉽습니다.

이 장의 목표는 스퀵클래스의 전체를 자세하게 들여다 보는것이 아니라, 프로그램 작성시 사용할 필요가 있거나 오버라이드할때 효과적으로 사용될 수 있는 중요한 클래스와 매서드를 살펴보는 겁니다: Object, Number 그리고 subclasses, Character, String, Symbol, `Boolean` 등이 이런 중요한 클래스들입니다.

8.1 Object

어떠한 의도 또는 목적에 대해서도 `Object` 는 상속 계층의 뿌리*root*(또는 최하위 계층) 가 됩니다. 실제로 스몰토크에서, 상속도의 진짜 최하위 계층은 ProtoObject이며, ProtoObject는 객체로서 가장된 최소한의 엔티티를 정

의하기 위해 사용되지만, 지금의 스몰토크는 이런 ProtoObject에 대한 내용을 크게 신경쓰지 않아도 됩니다.

Obejct는 kernel-Object 카테고리에서 발견할 수 있습니다. 놀랍게도, 여기에서는 400개의 메서드(확장을 포함한)를 발견할 수 있습니다. 달리 말한다면, 스몰토크에서 사용자가 정의하는 모든 클래스는 사용자가 이런 메서드들을 아는지와는 상관없이 400개의 메서드를 자동으로 제공한다는 의미가 됩니다. 몇몇메서드가 없을수도 있는데 새버전의 Squeak에서는 몇몇 필요없는 메서드를 제거할 수도 있기 때문이라는 것에 주의해야 합니다.

Object 클래스에 적혀있는 주석:

> *Object*는 클래스 계층도에 속한 대부분의 다른 클래스들을 위한 루트 클래스*the root class* 입니다. 예외가 있다면 `ProtoObject`*(Object*의 *superclass)*와 그것의 서브클래스들*subclases* 입니다. `Object` 클래스는 접근, 복사, 비교, 오류 처리, 메시지 전송, 그리고 반영과 같은 모든 일반적인 객체에 공통으로 적용되는 기본 동작을 제공합니다. 또한 모든 객체들이 응답해야할 유틸리티 메시지는 이 곳에서 정의됩니다. *Object*는 인스턴스 변수를 가지고 있지 않으며 그것*(*인스턴스변수*)*를 추가해야할 필요도 없습니다. 이런 특성은 *Object*의 여러 클래스들이 특별한 실행*(*예를 들어, `smallInteger` 와 `UndefinedObject`*)*을 가지는 `Object` 부터 상속되었기 때문이거나, 또는 *VM*이 특정한 표준 클래스의 구조와 레이아웃에 관해 알고있거나 그것들을 기반으로 하기 때문입니다.

만약 Object의 instance side에서 메서드 카테고리 탐색을 시작하면, 여기서 제공되는 몇가지 중요한 동작을 발견하기 시작할 수 있을겁니다.

출력 (Printing)

스몰토크에서 모든 객체들은 그 자체의 출력 폼을 반환할 수 있습니다. 사용자는 워크스페이스에서 모든 표현식을 선택할 수 있고 print it 메뉴를 선택할 수 있습니다: 이런 동작들은 프로그램식을 실행하고 반환된 객체에게 그 자체를

출력하도록 요청합니다. 사실 출력에 대한 동작은 반환된 객체에 printString 메시지을 전송합니다. 메서드 `printString` 은 템플릿 메서드이며 그 핵심은 메시지 `printOn:` 을 그 자신의 수신자(receiver)에 발송하는것에 있습니다. printOn:은 hook 메서드 로서 특별하게 취급됩니다.

`Object>>printOn:` 은 사용자가 가장 빈번하게 오버라이딩 하게되는 메서드중 하나가 될겁니다. `printOn` 메서드는 인자로 Stream 을 받은후, 받은 인자를 이용해서 String 으로 객체의 이름을 표현하는것을 만들어냅니다. printOn 의 기본적인 구현은 "`a`" 또는 "`an`" 뒤에 클래스 이름을 사용하는것 밖에 없습니다. `Object>>printString`은 이렇게 만들어진 문자열을 반환 합니다.[1]

예를 들어, 보면, 클래스 브라우저는 `printOn:` 메서드를 재정의 하지 않으며, printSting 메시지를 Object에서 정의된 메서드들을 실행하는 인스턴스로 전송합니다.

```
Browser new printString  ⟶  'a Browser'
```

`TTCFont` 클래스에서 `printOn:` 특수화에 대한 좋은 예를 확인할 수 있습니다. 아래의 코드에서 볼 수 있듯이 클래스의 인스턴스를 출력할때, 폰트의 모음 이름, 크기 그리고 종속 모음 이름이 클래스 이름의 뒤에 같이 출력됩니다.

Method 8.1: `printOn:` 메서드의 재정의

```
TTCFont>>printOn: aStream
        aStream nextPutAll: 'TTCFont(';
        nextPutAll: self familyName; space;
        print: self pointSize; space;
        nextPutAll: self subfamilyName;
        nextPut: $)
```

```
TTCFont allInstances anyOne printString  ⟶
    'TTCFont(BitstreamVeraSans 6 Bold)'
```

메시지 `PrintOn:`은 `storeOn:` 과 같지 않다는것에 주의해주세요. `storeOn:` 메시지는 수신자를 재생성하는데 사용할 수 있는 프로그램식을 그 자체의

[1] http://ta.onionmixer.net/wordpress/?p=215 페이지를 참고해주세요

(storeOn의) 인자에 stream 으로 대입합니다. 이렇게 만들어진 프로그램식은 readFrom 메시지를 사용해서 읽어들일때 처리됩니다. `pringOn:`은 단지 수신자의 텍스트 버전을 반환할 뿐입니다. 물론 텍스트로 만들어진 프로그램식이 수신자를 표현하는 자체 처리 프로그램식이 될 수도 있습니다.

표현식과 자체평가표현식 이라는 단어에 대해서 함수 프로그래밍에서, 프로그램식은 실행이 될 때 값을 반환합니다. 스몰토크에서, 메시지 (프로그램식) 들은 객체 (값*value*) 를 반환합니다. 그 자신이 스스로 값으로 인식되는 좋은 속성들을 가지는 객체들이 있습니다. 예를 들어, 객체 `true`의 값은 그 자신으로서 true 라는 객체가 됩니다. 이러한 객체들을 자체평가 객체*self-evaluating objects* 라 부릅니다. 워크스페이스에서 객체를 출력 (print it) 할 때 객체의 출력용 버전을 볼 수 있습니다. 여기에 위에서 설명한 자체평가 프로그램식의 예제가 있습니다.

```
true      ⟶  true
3@4       ⟶  3@4
$a        ⟶  $a
#(1 2 3)  ⟶  #(1 2 3)
```

배열과 같은 몇몇 객체는 자체평가 또는 포함하고 있는 객체에 의존하지 않는다는 사실을 주의해주세요. 예를 들어, `Boolean` 으로 만들어진 배열은 자체평가가 되지면 persons의 배열은 그렇지 않다는거죠. 스퀵3.9에서 이러한 동작방식은 가능한 많이 (`isSelfEvaluating` 메시지처럼) 자체평가 형태로 컬렉션을 출력하기 위해 도입되었으며, 이는 특별히 중괄호 배열에 있어 true가 됩니다. 다음 예제는 동적 배열의 구성요소들일 경우에만, 동적 배열이 자체처리됨을 보여드립니다. 다음 예제는 동적배열의 구성요소가 자체평가형인 경우 동적 배열이 자체평가로 처리됨을 보여줍니다:

```
{10@10 . 100@100}        ⟶  {10@10 . 100@100}
{Browser new . 100@100}  ⟶  an Array(a Browser 100@100)
```

리터럴 배열은 오직 리터럴만 포함할 수 있다는걸 기억해주세요. 그렇기 때문에 다음의 배열은 두 개의 점 (point) 을 가지지 않으며 여섯개의 리터럴 요소만을 가지게 됩니다.

```
#(10@10 100@100)  ⟶  #(10 #@ 10 100 #@ 100)
```

printOn: 메서드의 대부분은 자체평가를 구현하는 특수화를 진행하게 됩니다. Point>>printOn:과 Interval>>printOn:의 구현은 자체평가방식이 됩니다.

Method 8.2: Point의 자체평가

```
Point>>printOn: aStream
    "The receiver prints on aStream in terms of infix notation."
    x printOn: aStream.
    aStream nextPut: $@.
    y printOn: aStream
```

Method 8.3: Interval자체평가 *(Self-evaluation)*

```
Interval>>printOn: aStream
    aStream nextPut: $(;
        print: start;
        nextPutAll: ' to: ';
        print: stop.
    step ~= 1 ifTrue: [aStream nextPutAll: ' by: '; print: step].
    aStream nextPut: $)
```

```
1 to: 10  ⟶  (1 to: 10)    "intervals are self--evaluating"
```

정체성 (Identity) 과 동일성 (equality)

스몰토크에서 = 메시지는 객체가 같은지 (동일성) 를 테스트하며 (예 : 두개의 객체가 같은값을 가지는지에 대해), == 메시지는 객체의 정체성 (identity) 을 테스트 합니다 (예 : 두개의 프로그램식이 같은 객체를 나타내는지에 대한 부분)

객체 동일성 비교에 대한 기본구현은 객체의 정체성에 대한 테스트입니다.

Method 8.4: 객체의 동일성

```
Object>>= anObject
    "Answer whether the receiver and the argument represent the
    same object.
```

```
    If = is redefined in any subclass, consider also redefining
    the message hash."
    ↑ self == anObject
```

아래에 보이는것은 자주 오버라이드 하게될 메서드 입니다. 복소수의 경우를 생각해주세요:

```
(1 + 2 i) = (1 + 2 i)   ⟶  true     "same value"
(1 + 2 i) == (1 + 2 i)  ⟶  false    "but different objects"
```

이렇게 동작하는 이유는, Complex가 = 를 다음과 같이 오버라이드(override) 하고 있기 때문입니다.

Method 8.5: 복소수를 위한 동일성

```
Complex>>= anObject
    anObject isComplex
        ifTrue: [↑ (real = anObject real) & (imaginary = anObject
    imaginary)]
        ifFalse: [↑ anObject adaptToComplex: self andSend: #=]
```

Object~>>= 의 기본구현은 Object>>= 을 부정하는 의미가 됩니다. 일반적으로 변경해야할 필요는 없습니다.

```
(1 + 2 i) ~= (1 + 4 i)  ⟶  true
```

만약 = 을 오버라이드 한다면, hash 를 사용한 부분 또한 오버라이드 하지 않으면 안됩니다. 클래스의 인스턴스가 Dictionary의 키로서 사용되었을 경우에, "같다" 고 보이는 인스턴스는, hash값도 같아야 합니다:

Method 8.6: *hash*는 반드시 복소수등을 위해 재실행되어야 합니다.

```
Complex>>hash
    "Hash is reimplemented because = is implemented."
    ↑ real hash bitXor: imaginary hash.
```

비록 = 과 bash 를 함께 오버라이드 해야한다고 해도, == 를 재지정하면 안됩니다. (객체의 identity에 대한 의미는 모든 클래스에 동일합니다) == 기호는 ProtoObject 의 프리미티브 메서드 이기 때문입니다.

스퀵이 다른 스몰토크구현과는 다른 몇가지 익숙하지 않은 동작을 한다는것에 주의해주세요 : 예를들면 symbol 과 string은 동일하지 않습니다.(이건 기능이 아니라 버그가 아닌가 라는 생각을 하고있습니다)

```
#'lulu' = 'lulu'  ⟶  true
'lulu' = #'lulu'  ⟶  true
```

클래스 멤버십

몇몇의 메서드를 이용하면 사용자는 객체의 클래스를 조회할 수 있습니다.

`class`. class 라는 메시지를 사용하면 모든 객체에 대해 객체의 클래스가 무엇인지를 요청할 수 있습니다.

```
1 class  ⟶  SmallInteger
```

이와는 반대로, 객체가 특정 클래스의 인스턴스가 맞는지의 여부에 대한 확인을 요청할 수 있는 방법도 있습니다 :

```
1 isMemberOf: SmallInteger  ⟶  true    "must be precisely this
    class"
1 isMemberOf: Integer       ⟶  false
1 isMemberOf: Number        ⟶  false
1 isMemberOf: Object        ⟶  false
```

스몰토크는 스몰토크스스로를 이용해서 만들어져 있기 때문에, 사용자는 상위클래스와 메시지(12장 을 참고해주세요)의 정상적인 조합을 사용해서 스몰토크의 구조 전체를 탐색할 수 있습니다.

`isKindOf:` `Object>>isKindOf:` 는 수신자의 클래스가 인자클래스와 동일한지, 또는 서브클래스인지등을 반환합니다.

```
1 isKindOf: SmallInteger  ⟶  true
1 isKindOf: Integer       ⟶  true
1 isKindOf: Number        ⟶  true
1 isKindOf: Object        ⟶  true
1 isKindOf: String        ⟶  false

1/3 isKindOf: Number      ⟶  true
1/3 isKindOf: Integer     ⟶  false
```

1/3 은 분수인 동시에 숫자의 한 종류이며, Integer 클래스는 분수 클래스의 super클래스이지만, 1/3 이 정수가 되는것은 아닙니다.

respondsTo: Object>>respondsTo: 는 수신자의 클래스가 인자의 클래스와 동일한지, 또는 서브클래스인지에 대한 정보등을 반환합니다.

```
1 respondsTo: #,  ⟶  false
```

일반적으로 객체에 클래스를 질의하거나, 객체가 어떤 메시지를 이해할 수 있는지에 대해서 객체에게 요청하는 것은 좋은 방법이 아닙니다. 객체의 클래스에 근거해서 판단하는 대신, 객체에 메시지만을 보낸후 수신한 객체가 어떤 행동을 할지는 객체의 결정(예를들자면 객체의 클래스에 기반한)에 맡겨야 합니다.

객체의 복사

객체를 복사하는 것은 몇 가지 미묘한 문제들을 발생시킵니다. 인스턴스 변수들은 참조에 의해 접근되기 때문에, 객체의 얕은 복사는 인스턴스 변수에 대한 그 객체의 참조를 원본 객체와 공유합니다:

```
a1 := { { 'harry' } }.
a1  ⟶  #(#('harry'))
a2 := a1 shallowCopy.
a2  ⟶  #(#('harry'))
(a1 at: 1) at: 1 put: 'sally'.
a1  ⟶  #(#('sally'))
a2  ⟶  #(#('sally'))    "the subarray is shared!"
```

Object>>shallowCopy 는 객체의 얕은 복사를 만드는 프리미티브 메서드입니다. a2가 a1의 유일한 얕은 복사이기 때문에, 두 개의 배열은 공통으로 겹치는 배열에 대한 참조를 공유합니다.

Object>>shallowCopy는 Object>>copy 에 대한 "공용 인터페이스" 이며 만약 인스턴스들이 고유한 경우 반드시 재지정(override) 해야만 합니다. 예를 들어, Boolean, Character, SmallInteger, Symbol 클래스, UndefinedObject 등이 이 경우에 해당됩니다.

`Object>>copyTwoLevel` 은 간단한 얕은 복사로 잘 되지 않을때, 정확한 작업을 진행합니다:

```
a1 := { { 'harry' } } .
a2 := a1 copyTwoLevel.
(a1 at: 1) at: 1 put: 'sally'.
a1  ⟶  #(#('sally'))
a2  ⟶  #(#('harry'))    "fully independent state"
```

`Object>>deepCopy` 는 임의로 객체의 깊은 복사를 만듭니다.

```
a1 := { { { 'harry' } } } .
a2 := a1 deepCopy.
(a1 at: 1) at: 1 put: 'sally'.
a1  ⟶  #(#('sally'))
a2  ⟶  #(#(#('harry')))
```

깊은 복사의 문제는 상호 재귀적 구조에 사용되는경우, 종료되지 않는다는 것입니다:

```
a1 := { 'harry' }.
a2 := { a1 }.
a1 at: 1 put: a2.
a1 deepCopy  ⟶  !\emph{... does not terminate!}!
```

올바른 작동을 위해 deepCopy의 재지정(override)을 할 수도 있지만, 보통 `Object>>copy` 가 더 나은 방법이라고 할 수 있습니다:

Method 8.7: 템플릿 메서드로 객체를 복사하기

```
Object>>copy
    "Answer another instance just like the receiver. Subclasses
    typically override postCopy;
    they typically do not override shallowCopy."
    ↑self shallowCopy postCopy
```

공유되지 않아야 할 모든 인스턴스 변수들을 복사하기 위해 여러분은 반드시 postCopy를 재지정해야 합니다. posetCopy는 반드시 super postCopy를 실행해야 합니다.

디버깅

이 부분에서 가장 중요한 메서드는 halt 입니다. 메서드에서 중단점 (breakpoint) 을 설정하기 위해, 메서드의 내용중 원하는 부분에 메세지를 보내는 "self halt" 를 삽입합니다. 메시지가 보내질 때, 실행은 중단되며, 디버거는 프로그램내의 중단된 지점에서 열리게 됩니다. (디버거에 관한 좀 더 자세한 내용은 6장 을 보십시오.)

다음으로 중요한 메시지는 assert: 이며, 이 메시지는 블록을 인자로 요구합니다. 만약 블록이 true 를 반환하면 계속 실행됩니다. true가 아닌경우, AssertionFailure 예외가 발생됩니다. 만약 이 예외를 인지하지 못하면, 예외가 발생한 지점에서 디버거가 열립니다. assert: 메시지는 명세에 의한 설계를 진행하는 작업을 지원해야하는 작업등에 유용합니다. 객체의 공용 메서드에 대한 중요한 사전조건을 점검하는 작업등에서 사용되는경우가 일반적입니다. `Stack>>pop`는 다음처럼 쉽게 구현되어 있습니다:

Method 8.8: 전제조건 점검

```
Stack>>pop
    "Return the first element and remove it from the stack."
    self assert: [ self isEmpty not ].
    ↑self linkedList removeFirst element
```

`TestCase>>assert:`와 `Object>>assert:` 을 혼돈하지 마십시오, `Object>>assert`은 SUnit 테스트 프레임워크에서 발견할 수 있습니다 (7장 을 보십시오). 전자가 인자[2] 로서 블록을 필요로하는 반면에, 후자는 `Boolean` 을 인자로서 받습니다. 하지만, 두가지 모두 디버깅에 유용합니다만, 사용목적은 크게 다릅니다.

오류 처리

이 프로토콜은 런타임 오류를 나타내는데에 유용한 여러 개의 메서드들을 포함하고 있습니다.

[2] 실제로, 이것은 `Boolean`을 포함한 값을 이해하는 모든 인자를 취할 것입니다.

설정 브라우저 (Preference browser) 에서 deprecation 을 활성한 경우에 self deprecated: anExplanationString 신호를 보내면 현재 메서드는 더이상 사용되지 않는것이 되며, String 으로 반환되는 인자는 대안을 제시하게 됩니다.

```
1 doIfNotNil: [ :arg | arg printString, ' is not nil' ]
     ⟶  !\emph{SmallInteger(Object)>>doIfNotNil: has been
    deprecated. use ifNotNilDo:}!
```

`doesNotUnderstand:` 는 메시지 lookup에 실패할 때마다 전송됩니다. 기본 구현은 그렇습니다만, 예를 들자면 `Object>>doesNotUnderstand:`는 이 시점에서 디버거를 열게됩니다. 이 외에 다른 동작을 제공하려면 doestNotUnderstand: 재지정 하는것이 좋습니다.

`Object>>error` 와 `Object>>error:` 는 예외를 발생시키기 위해 사용될 수 있는 범용적인 메서드입니다. (하지만 대부분의 경우 kernel 클래스와 코드에서 발생하는 오류를 구분할 수 있도록 사용자가 별도로 예외를 발생시키는것이 좋습니다.)

스몰토크에서 추상 메서드는 규칙에 따라 `self subclassResponsibility` 의 본문에 구현되어 있습니다. 따라서 추상클래스가 우연히 (실수로) 인스턴스화 되어 호출되어도 `Object>>subclassResponsibility` 가 실행됩니다.

Method 8.9: 추상 메서드임을 신호로 알리기

```
Object>>subclassResponsibility
    "This message sets up a framework for the behavior of the
    class' subclasses.
    Announce that the subclass should have implemented this
    message."
    self error: 'My subclass should have overridden ',
    thisContext sender selector printString
```

Magnitude, Number, `Boolean` 은 이 장에서 잠깐 살펴볼 추상 클래스들의 고전적인 예입니다.

```
Number new + 1  ⟶  !\emph{Error: My subclass should have
    overridden \#+}!
```

관례에 따라 서브클래스가 상속해서는 안되는 메서드를 알려주려면 서브클래스쪽에서 `self ShouldNotImplement` 를 보냅니다. 이건 일반적인 경우 클래스 계층의 설계가 잘못된 경우의 신호(signal) 입니다. 그렇지만 단일상속에서 오는 한계때문에, 가끔 이런 예비수단을 사용해야만 하죠.

대표적인 예로서, 딕셔너리에 의해 상속되었지만 실행된 것으로 알려지지 않은 `Collection>>remove:` 가 있습니다. (딕셔너리는 `removeKey:` 를 제공합니다.)

Testing

testing 메서드는 SUnit 테스트와는 아무 관계가 없습니다. testing 메서드는 여러분에게 수신자*receiver* 의 상태에 대한 질문을 하고 `Boolean`(`Boolean`) 을 반환하는 메서드입니다.

`Object`는 여러개의 testing 메서드를 제공합니다. `isComplex`는 이미 확인되었습니다. 그 외에도 `isArray`, `isBoolean`, `isBlock`, `isCollection` 등이 있습니다. 일반적으로 이러한 메서드들은 가능하면 사용하지 말아야 합니다. 그 이유는 메서드의 클래스를 위해 객체를 조회*quering* 하는 작업은 캡슐화*encapsulation* 에 위배*a form of violation* 되기 때문입니다. 사용자는 객체의 클래스를 테스트 하는 대신에, 요청*request* 을 보내서 객체로 하여금, 그 클래스를 어떻게 제어할지 결정하도록 해야합니다.

그렇지만, 이 테스트 메서드중 일부는 정말로 유용합니다. 가장 유용한 것은 아마도 `ProtoObject>>isNil` 과 `Object>>notNil` 일 것입니다.(Null 객체 [6] 의 디자인 패턴을 사용하면 이런 메서드가 필요한 경우를 없앨 수 있습니다.)

initialize release

Object가 아닌 ProtoObject에 잇는 마지막 중요한 메서드는 initialize 입니다.

Method 8.10: 빈 *hook* 메서드로서의 initialize

```
ProtoObject>>initialize
```

```
    "Subclasses should redefine this method to perform
    initializations on instance creation"
```

squeak 3.9 버전에서 이게 중요한 이유는, 기본적으로 시스템의 모든 클래스에서 new 메서드는 새로 생성된 인스턴스에 initialize 를 보내게 되기 때문입니다.

Method 8.11: 클래스-사이드 템플릿 메서드로서의 new

```
Behavior>>new
    "Answer a new initialized instance of the receiver (which is
    a class) with no indexable
    variables. Fail if the class is indexable."
    ↑ self basicNew initialize
```

위의 내용은 initialize hook 메서드를 단순히 재지정하면, 클래스의 인스턴스가 자동으로 초기화된다는 의미입니다.

일반적으로 initialize 메서드는 상속된 모든 인스턴스 변수들을 위한 클래스 불변성을 확보하기 위해 super initialize를 수행합니다.(이런 동작이 다른 스몰토크에서 표준은 아니라는것을 주의해주세요)

8.2 Number

놀랍게도, 스몰토크에서의 숫자들은 원시데이터가 아닌 진짜 객체 입니다. 물론 숫자들은 가상 머신에서 효과적으로 동작합니다만, `Number` 의 계층은 스몰토크클래스 계층의 다른 부분들처럼 완전하게 접근 및 확장이 가능합니다.

Number(숫자형)과 관련된 내용은 kernel-number 카테고리에서 찾을 수 있습니다. 이 계층의 상위는 Magnitude이며, Magnitude는 비교 연산자를 지원하는 모든 종류의 클래스들을 의미합니다. Number는 다양한 산술 요소를 추가할 수 있습니다. 이런 연산자의 대부분이 추상메서드 입니다. Float와 Fraction은 각각 부동소수점(실수)과 분수를 나타냅니다. Integer 또한 추상클래스 이며, 추상클래스인 Integer는 하위 클래스인 SmallInteger, Lage-

PositiveInteger, LargeNegativeInteger 등과는 다르게 구별됩니다. 일반적인 경우, 값이 요구하는 만큼 자동으로 변환되므로, 유저들은 3 개의 정수 클래스들 사이의 차이점들에 대해 신경을 써야 할 필요는 없습니다.

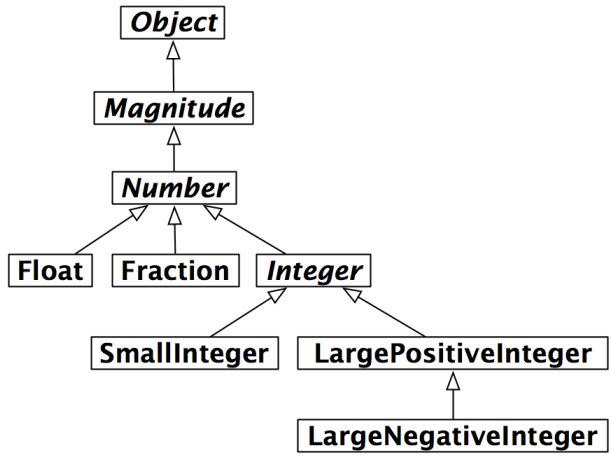


그림 8.1: 숫자구조도 (class구조)

Magnitude

Magnitude는 Number 클래스의 부모클래스일 뿐만 아니라 Character, Duration, Timespan 처럼 연산을 지원하는 클래스들의 부모이기도 합니다.(Complex(복소수)는 비교가 불가능합니다. 그래서 Number 를 상속받지는 않습니다)

<과 = 메서드는 추상메서드입니다. 그외 다른 연산자들은 일반적인 방법으로 정의됩니다. 예를 들자면 다음과 같습니다:

Method 8.12: 추상 비교 메서드

```
Magnitude>> < aMagnitude
    "Answer whether the receiver is less than the argument."
    ↑self subclassResponsibility

Magnitude>> > aMagnitude
    "Answer whether the receiver is greater than the argument."
    ↑aMagnitude < self
```

Number

Magnitude 비슷하게 `Number` 에서 `+`, `-`, `*`, `/` 등은 추상메서드로 정의합니다만, 그외의 다른 산술 연산자들은 일반형적 형태로 정의됩니다.

모든 Number 객체들은 asFloat 과 asInteger 같은 다양한 변환 연산자를 지원합니다. 이런 연산자중에는, Number 를 실수부가 0 인 복소수의 인스턴스로 변환하는 i나, Number에서 Duration 객체를 생성하는 hour, day, week 등이 있습니다.

Number는 sin, log, raiseTo:, squared, sqrt 등과 같은 일반 수학 기능을 직접 지원합니다.

`Number>>printOn:`은 `Number>>printOn:base:`의 추상메서드로 구현되어 있습니다(`Number>>printOn:`의 경우 base에 해당되는 기본값은 10으로 지정되어 있습니다.)

Testing 메서드의 종류에는, even, odd, positive, negative 등이 있습니다. 당연히, Number는 isNumber를 재지정(override)하고 있습니다. 재미있는 점은 isInfinite는 false 를 반환하도록 정의되어 있다는 것입니다.

Truncation 메서드에는 floor, ceiling, intergerPart, fractionPart 등이 있습니다.

```
1 + 2.5    ⟶  3.5    "Addition of two numbers"
3.4 * 5    ⟶  17.0   "Multiplication of two numbers"
8 / 2      ⟶  4      "Division of two numbers"
10 -- 8.3  ⟶  1.7    "Subtraction of two numbers"
12 = 11    ⟶  false  "Equality between two numbers"
12 ~= 11   ⟶  true   "Test if two numbers are different"
12 > 9     ⟶  true   "Greater than"
12 >= 10   ⟶  true   "Greater or equal  than"
12 < 10    ⟶  false  "Smaller than"
100@10     ⟶  100@10 "Point creation"
```

다음 예제는 스몰토크에서 문제없이 잘 작동합니다:

```
1000 factorial / 999 factorial  ⟶  1000
```

1000 factorial 을 다른언어로 계산하는건 쉽지 않습니다만 별다른 어려움없이 계산이 된다는점을 주의해 주시기 바랍니다. 위의 예제는 숫자에 대한 자동 형변환과 정확한 숫자의 정확한 제어에 대한 좋은 예입니다.

 `1000 factorial`의 결과 출력을 시도해 보시기 바랍니다. 이 작업은 계산하는 것보다 출력에 더 많은 시간이 걸립니다.

Float

Float는 부동소수점을 위한 Number의 추상메서드로 구현되어 있습니다.

여기서 흥미로운점은 Float 클래스 (예컨데 Float의 class-side) 는 다음 정수를 반환합니다 : e, infinity, nan, pi 등

```
Float pi                    ⟶  3.141592653589793
Float infinity              ⟶  Infinity
Float infinity isInfinite   ⟶  true
```

Fraction

Fraction(분수) 은 분자와 분모에 쓰이는 인스턴스 변수들로 표시되며, 각 분자와 분모는 반드시 정수여야 합니다. 분수는 일반적으로 정수의 나눗셈으로 만들어 집니다. (`Fraction>>numerator:denominator:` 를 사용하는것 보다는 constructor 메서드가 낫습니다.)

```
6/8          ⟶  (3/4)
(6/8) class  ⟶  Fraction
```

정수 또는 다른 분수로 분수를 곱하기 하면, 답이 정수가 될 수 있습니다.

```
6/8 * 4  ⟶  3
```

Integer

`Integer정수()` 는 3 개의 구체적 정수 구현에 대한 추상적 부모 클래스입니다. Integer는 많은 추상 Number 메서드의 구체적인 구현을 제공할 뿐만 아니

라, factorial, atRandom, isPrime, gcd 등의 정수 특유의 메서드 및 그외 많은 메서드를 추가합니다.

SmallInteger는 인스턴스가 간결하게 표시된다는 면에서 특별합니다-SmallInteger는 값을 참조로서 저장하는 대신, 참조를 유지하기 위해 비트 영역을 사용해 직접 표현합니다. 객체참조의 첫 번째 비트는 객체가 SmallInteger인지 아닌지를 가리킵니다.

클래스 메서드 minVal 과 maxVal 은 SmallInteger의 범위를 알려줍니다.

```
SmallInteger maxVal = ((2 raisedTo: 30) -- 1)    ⟶  true
SmallInteger minVal = (2 raisedTo: 30) negated  ⟶  true
```

SmallInteger가 이 범위를 벗어나면, LargePositiveInteger 로 자동으로 변환되거나 또한 필요한 만큼 LargeNegativeInteger 로 자동으로 변환됩니다:

```
(SmallInteger maxVal + 1) class  ⟶  LargePositiveInteger
(SmallInteger minVal -- 1) class  ⟶  LargeNegativeInteger
```

마찬가지로, 큰 정수들은 알맞은 시기에 작은 정수들로 다시 변환됩니다.

대부분의 프로그래밍 언어에서 처럼, 정수들은 지정동작의 반복에 유용합니다. 블록을 반복적으로 평가하는 작업에 사용되는 전용 메서드 timesRepeat: 이 있습니다. 3장 에서 비슷한 예제를 이미 본적이 있죠.

```
n := 2.
3 timesRepeat: [ n := n*n ].
n  ⟶  256
```

8.3 Character

Character 는 Magnitude 의 하위 클래스로서 Collections-String 에 정의되어 있습니다. 표시할 수 있는 문자는 $⟨char⟩ 처럼 스퀵에서는 표현됩니다. 예를 들어, 이렇게 말이죠:

```
$a < $b  ⟶  true
```

인쇄-불가능한 문자는 다양한 클래스 메서드들을 통해서 발생됩니다. `Character class>>value:`는 Unicode (또는 ASCII) 정수 값을 인수로서 사용하고 해당하는 문자를 반환합니다. `accessing untyPeable character` 프로토콜은 `backspace`, `cr`, `escape`, `euro`, `space`, `tab` 등과 같은 여러가지 편리한 생성자 메서드를 포함하고 있습니다.

```
Character space = (Character value: Character space asciiValue)
      ⟶  true
```

`printOn:` 메서드는 문자를 표현하는 세가지 방법중 어떤것이 가장 적합한지를 판단할 수 있을만큼 똑똑합니다.:

```
Character value: 1   ⟶  Character value: 1
Character value: 32  ⟶  Character space
Character value: 97  ⟶  $a
```

`isAlphaNumeric`, `isCharacter`, `isDigit`, `isLowercase`, `isVowel` 와 같은 다양하고 편리한 *testing* 메서드가 내장되어 있습니다.

`Character`를, 바로 그 문자를 포함한 문자열로 변환하시려면 `asString` 을 전송합니다. 이번 경우, asString 과 `pringString` 은 다른 결과를 보여줍니다:

```
$a asString     ⟶  'a'
$a              ⟶  $a
$a printString  ⟶  '$a'
```

모든 ASCII 문자는 클래스 변수 CharacterTable에 저장된 고유한 인스턴스입니다:

```
(Character value: 97) == $a  ⟶  true
```

하지만 0 에서 255 범위 밖의 character들은 고유하지 않습니다. 아래와같이 말이죠[3]:

```
Character characterTable size                      ⟶  256
(Character value: 500) == (Character value: 500)   ⟶  false
```

[3]예제중 두번째것을 보면 당연히 같은 500값에 해당하므로 비교시 true가 출력되어야 하나 false가 출력되고 있습니다. 그래서 ASCII 를 벗어나는 문자는 동일하지 않을 수 있다고 설명하는 것이죠.

8.4 String

String 또한 Collection-String 카테고리에 정의되어 있습니다. 하나의 String 은 오직 character 들만 가지고 있는 색인화된 Collection 입니다.

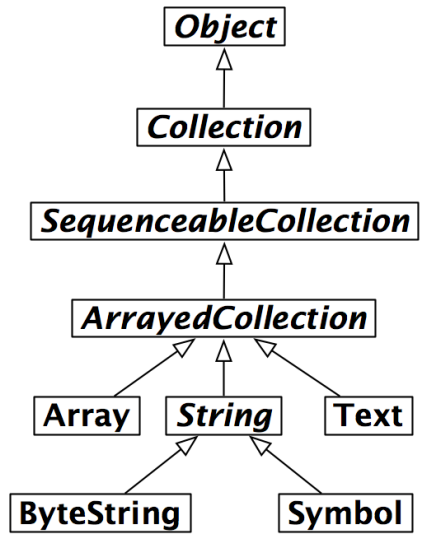


그림 8.2: 문자열 계층도

사실, String은 추상클래스이며, 스퀵의 String은 사실상 실제하는(concrete) 클래스 ByteString의 인스턴스입니다.

```
'hello world' class  ⟶  ByteString
```

다른 중요한점은 String의 서브클래스는 Symbol이라는 것입니다. 중요한 차이점은 Symbol의 인스턴스는 하나의 값에 대해 하나만 존재하다는 점이죠.(이는 때때로 "고유 인스턴스 속성" 으로 불리기도 합니다.) 한편 개별로 생성되었지만 우연히 같은 문자열을 가지는 String은 서로 다른 객체인 경우도 종종 있습니다.

```
'hel','lo' == 'hello'  ⟶  false
```

```
('hel','lo') asSymbol == #hello  ⟶  true
```

또다른 중요한 차이점은 Symbol은 변경이 불가능하지만, String은 변경이 가능하다는 점입니다.

```
'hello' at: 2 put: $u; yourself  ⟶  'hullo'
```

```
#hello at: 2 put: $u  ⟶  error!
```

문자열들이 Collection이라는 사실을 잊어버리기가 쉽습니다. 문자열들은 Collection 들이 이해하는것과 동일하게 메시지를 이해합니다:

```
#hello indexOf: $o  ⟶  5
```

비록 String이 Magnitude 에서 상속된 것은 아닙니다만, 일반적으로, 비교 메서드인 <, =등을 지원합니다. 게다가, String>>match 는 기본적인 glob-style 패턴 비교에 유용합니다.

```
'*or*' match: 'zorro'  ⟶  true
```

정규 표현식에 사용할만한 보다 나은 지원이 필요하다면, Vassili Bykov의 Regex 패키지같은 여러 가지 서드파티등을 활용할 수 있습니다.

문자열은 여러가지 변환 메서드들을 지원합니다. 이 메서드 중 대부분은 asDate, asFileName 등과 같은 다른 클래스들을 위한 shortcut 생성자 메서드입니다. capitalized 와 translateToLowercase 같이 문자열을 다른 문자열로 변환시키는 여러가지 유용한 메서드들도 있습니다.

문자열과 컬렉션에 대한 좀 더 많은 정보를 원하시면, 9장 을 보십시오.

8.5 Boolean

Boolean 클래스는 스몰토크언어가 얼마나 클래스 라이브러리에 강한 영향을 받는지에 대한 매우 흥미로운 예제입니다. Boolean은 singleton 으로 구현한 클래스인 True와 False의 추상적 상위클래스입니다.

Boolean 의 대부분의 동작에 대해서는 ifTrue:ifFalse: 메서드를 자세히 살펴보면 이해할 수 있으며, 이 메서드는 인수로서 두 개의 블록을 요구합니다.

```
(4 factorial > 20) ifTrue: [ 'bigger' ] ifFalse: [ 'smaller' ]
     ⟶  'bigger'
```

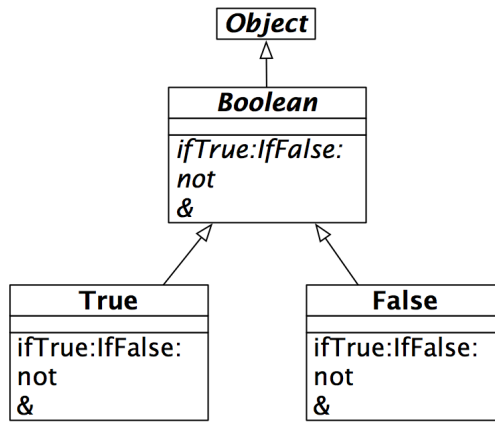


그림 8.3: Boolean 클래스 계층도

이것의 `concrete subclasses`에서의 실행은 모두 사소한 것들입니다.

이 메서드는 `Boolean` 안의 추상메서드 입니다. 서브클래스에서 구현되어있는 내용은 대단히 간단합니다:

Method 8.13: ifTrue:ifFalse:의 실행

```
True>>ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
    ↑trueAlternativeBlock value

False>>ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
    ↑falseAlternativeBlock value
```

사실 이런것들이 OOP의 진수입니다: 메시지를 객체에 전송할때, 객체 스스로 수신받은 메세지에 따라 어떤 메서드를 사용할지를 결정하게 됩니다. 이 경우, False의 인스턴스가 false의 선택사항을 결정하며, True의 인스턴스는 true의 선택사항을 결정합니다. `Boolean` 의 모든 추상메서드는 True와 False가 이러한 방법으로 구현되고 있습니다. 예를 들어, 보면 아래와 같습니다:

Method 8.14: *negation* 실행하기

```
True>>not
    "Negation----answer false since the receiver is true."
    ↑false
```

Boolean은 ifTrue:, ifFalse:, ifFalse:ifTrue 와 같은 유용하고 편리한 메서드들을 제공합니다. 그리고 eager와 lazy conjunction과, disjunctions 사이에서 선택도 가능합니다.

```
(1>2) & (3<4)              ⟶  false    "must evaluate both sides"
(1>2) and: [ 3<4 ]         ⟶  false    "only evaluate receiver"
(1>2) and: [ (1/0) > 0 ]   ⟶  false    "argument block is never
    evaluated, so no exception"
```

첫 번째 예제에서, & 양 옆의 Boolean 부분식 처리됩니다. 두 번째와 세 번째 예제에서는 and: 메서드가 인수로서 블록을 요구하기 때문에 결과적으로 좌측(수신자-receiver)만 처리됩니다. 우측(블록)은 첫 번째 인수가 true 인 경우에만 평가됩니다.

 and:와 or:가 어떻게 처리되는지를 예상해 보시기 바랍니다. Boolean에서 구현된 True와 False에서 실행*(implementations)*을 점검해 보시기 바랍니다.

8.6 8장 요약

- =(equal)를 재지정(override)한다면 hash 또한 재지정해야 합니다.
- 객체복사를 정확하게 구현하려면 postCopy를 재지정합니다.
- 중단점(breakpoint)을 설정하려면 self halt를 보냅니다.
- 추상 메서드를 만들려면 self subclassResponsibility를 반환합니다.
- 객체에 문자열(string) 표현을 부여하려면 printOn: 을 재지정해야 합니다.

- 인스턴스를 적당하게 초기화 하기 위해 hook 메서드인 initialize 를 재지정합니다.
- `Number` 클래스의 메서드는 자동으로 `Float`, `Fraction`, `Integer` 사이의 변환을 수행합니다.
- `Fraction`는 부동소수점수가 아닌 진짜 유리수를 나타냅니다.
- `Chracters`는 고유한 인스턴스입니다.
- `Strings`은 변경이 가능합니다만, `Symbol`은 변경할 수 없습니다. 문자열 리터럴을 변경할 수 없다는 것에 주의해 주십시오.
- `Symbol`은 고유하지만 `Strings`는 그렇지 않습니다.
- `Strings`와 `Symbol`은 컬렉션이므로 일반적인 컬렉션 메서드를 지원합니다.

제 9 장

컬렉션

9.1 개요

Collection 클래스는 컬렉션 (collection) 과 스트림 (stream) 의 범용의 서브클래스에 대략적으로 정의된 그룹입니다.

"Blue Book" [7] 에는 이러한 그룹의 구성으로 Collection의 17 개의 서브클래스와 Stream의 9개의 서브클래스를 합친, 총 28개의 클래스가 취급되고 있으며, Smalltalk-80 시스템이 릴리즈 되기 전에 이미 여러번 다시 설계되었습니다. 이런 클래스의 그룹은 객체지향 설계의 좋은예로 자주 언급됩니다.

스퀵에서, 추상 클래스인 `Collection`은 98개의 서브클래스를 갖고 있으며, 추상 클래인 Stream은 39 개의 서브 클래스를 갖고 있습니다만, 이 서브클래스들 중 대부분의 클래스는 (Bitmap, FileStram, CompiledMethod 등) 시스템의 다른 부분들 또는 어플리케이션에서 사용하기 위해 공들여 만든 특별한 목적에 쓰이는 클래스들입니다. 이 장의 목적에 맞춰, 컬렉션과 Collections-* 로 라벨이 붙은 시스템 카테고리에 있는 컬렉션의 37 개의 서브클래스들을 구분하기 위해 "컬렉션 계층" 이라는 용어를 사용하겠습니다. 또한 컬렉션 스트림 시스템 카테고리에 있는 Stream 과 그 Stream 으로 분류되는 10 개의 서브클래스들을 의미하기 위해 "스트림 계층" 이라는 용어를 사용하겠습니

다. 그림 9.1 에서 전체 목록을 확인할 수 있습니다. 이 49 개의 클래스들은 794 개의 메시지들에 반응하며, 총 1236 개의 메소드를 정의하고 있습니다.

이 장에서는, 그림 **??** 에서 볼 수 있는 컬렉션 클래스의 subset에 주로 집중할 것입니다. stream은 별도로 10장 에서 다루도록 하겠습니다.

9.2 컬렉션의 다양성

컬렉션 클래스를 보다 잘 사용하기 위해, 사용자는 적어도 컬렉션 클래스들의 외형 및 컬렉션 클래스들의 공통점들과 차이점들에 대한 지식을 알고 있는것이 좋습니다.

각각의 구성요소들보다는, 컬렉션들을 사용하여 프로그래밍을 하는 것이 프로그램의 추상성의 수준을 끌어올릴 수 있는 보다 중요한 방법입니다. Lisp 언어의 `map` 함수는 함수와 목록을 인자로 받아서 목록의 각 요소에 각각 함수를 적용해서 결과를 목록으로 반환하며, Smalltalk-80 은 핵심원리로 (이런 스타일을 따라) 컬렉션 기반 프로그래밍을 채택하였으며, ML 과 Haskell 과 같은 현대 함수기반 프로그래밍 언어들은 스몰토크의 스타일를 따랐다고 할 수 있습니다.

왜 이런것이 좋은 아이디어 일까요? 예를 들어, 학생에 대한 레코드의 집합인 데이터 구조가 있고, 몇가지 기준에 부합하는 모든 학생의 레코드에 대해 같은 동작을 진행하기 원한다고 하죠. 기존의 언어를 사용하는 프로그래머라면 반복문을 바로 생각하겠지만, 스몰토크프로그래머는 다음처럼 표현식 을 작성할 것입니다.

```
students select: [ :each | each gpa < threshold ]
```

위의 표현식은 대괄호 안의 함수가 `true`[1] 를 반환할때 students의 element를 포함하는 새 컬렉션으로 평가(evaluates)합니다. 스몰토크코드는 도메인-특화 조회 언어의 단순성과 우아함을 갖고 있습니다.

[1]대괄호에 있는 표현식은 익명의 함수 `_x.x gpa < threshold` 를 정의하는 표현식으로서 간주될 수 있습니다.

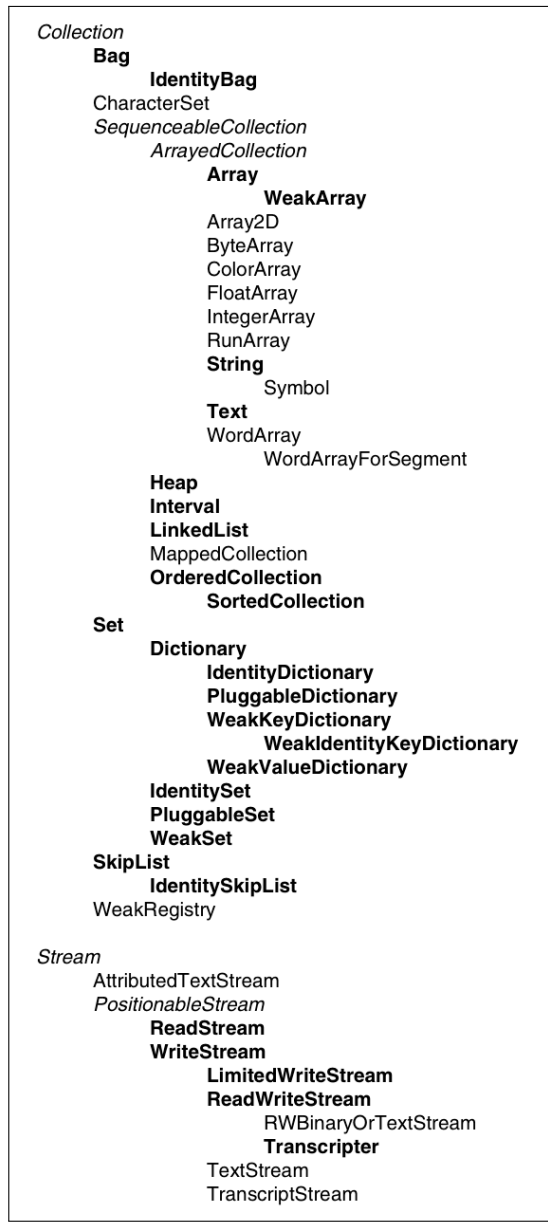


그림 9.1: 스퀵에서의 Collection 클래스. 들여쓰기(indent)는 하위분류를 나타냅니다. 이탤릭체로 쓴 클래스는 추상 클래스입니다. 강조체로 쓴 클래스는 "Blue Book"에 기술되어 있습니다.

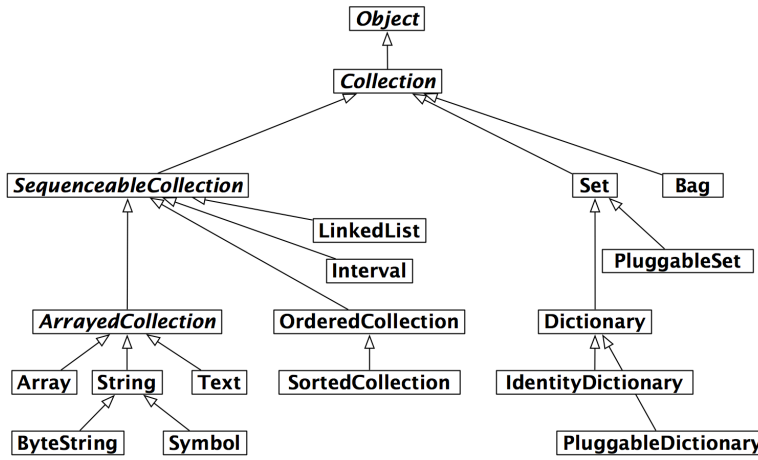


그림 9.2: 스퀵에서의 몇 가지 핵심 컬렉션 클래스

프로토콜	메서드
접근	size, capacity, at: anIndex, at: anIndex put: anElement
검사	isEmpty, includes: anElement, contains: aBlock, occurrencesOf: anElement
추가	add: anElement, addAll: aCollection
삭제	remove: anElement, remove: anElement ifAbsent: aBlock, removeAll: aCollection
열거	do: aBlock, collect: aBlock, select: aBlock, reject: aBlock, detect: aBlock, detect: aBlock ifNone: aNoneBlock, inject: aValue into: aBinaryBlock
변환	asBag, asSet, asOrderedCollection, asSortedCollection, asArray, asSortedCollection: aBlock
생성	with: anElement, with:with:, with:with:with:, with:with:with:with:, withAll: aCollection

그림 9.3: 표준 컬렉션 프로토콜

메시지 select:는 스몰토크의 모든 컬렉션에 전달될 수 있습니다. students의 데이터 구조가 배열 또는 linked list 인지를 확인할 필요는 없습니다: select: 메시지는 양쪽 모두에 전달될 수 있습니다. 반복(loop)를 사용하는경우에는 students가 배열인지 linked list 인지를 미리 알아두어야 합니다만, 반복과는 다르다는것에 주의해주시기 바랍니다. 스몰토크에서, 특정 컬렉션을 지칭하지않고 그냥 컬렉션 이라고만 하는 경우는, element가 대상이 되는 객체에 들어있는지를 테스트하고 열거하기위한 잘 정의된 프로토콜을 지원하는 객체를 의미합니다[2]. 모든 스몰토크의 컬렉션은 isEmpty와 occurrencesOf: 을 포함한 테스팅 메시지를 사용할 수 있습니다. 또한 모든 컬렉션은 열거 메시지 do:, select:, reject: (select:의 반대되는 것), collect:(Lisp의 map 과 같은), detect:ifNone:, inject:into: (왼쪽 fold를 수행하는)과 더 많은 메시지를 사용할 수 있습니다. 이런 프로토콜의 범용성과 다양성은 컬렉션을 강력하게 만들어주는 특징입니다.

그림 9.3 을 통해 컬렉션 계층의 대다수 클래스들이 지원하는 표준 프로토콜을 정리했습니다. 이 메서드들은 `Collection` 을 상속받은 서브클래스 내에서 정의(define), 재 정의(redefine), 최적화(optimized) 도 가능하지만, 가끔(서브클래스를 통한)사용이 금지되기도 합니다.

기본적인 일관성을 넘어선, 다양한 프로토콜들을 지원하거나 같은 요청에 다른동작을 제공하는 다수의 다른 컬렉션이 있습니다. 이제부터 중요한 몇가지 차이점들에 대해 알아보도록 하겠습니다:

순차가능(Sequenceable) SequenceableCollection의 모든 서브 클래스의 인스턴스는, first 요소로부터 시작되어, last 요소까지 명확하게 정해진 순서로 진행됩니다(sequenceable). 반대되는 경우로서, Set, Bag, Dictionary 의 인스턴스는 순서대로 배치할 수 없습니다.

분류가능(Sortable) SortedCollection은 해당 요소들을 정렬 순서에 따라 유지합니다.

[2] c++의 경우는 iterator가 이와 비슷합니다.

색인가능 (Indexable) 대부분의 sequenceable 콜렉션은 index 를 생성할 수 있으며, at: 을 이용해서 요소를 검색할 수 있습니다. Array 는 고정된 크기를 가진 친숙한 indexable 데이터 구조이며, `anArray at: n` 은 `anArray`의 `n`번째 (n^{th}) 요소를 검색하고, `anArray at: n put: v`는 `n` 번째의 요소의 값을 v 로 바꿉니다. LinkedLists와 SkipLists은 sequenceable이지만, indexable은 아닙니다. 즉 first와 last에는 반응할 수 있습니다만, at: 메서드는 이해하지 못합니다.

Keyed Dictionary와 그 서브클래스의 인스턴스는, 색인 대신에 키 (keys) 로 참조할 수 있습니다.

수정가능 (Mutable) 대부분의 컬렉션들은 변경이 가능하지만, Intervals와 Symbols는 그렇지 않습니다. Interval은 정수의 범위를 나타내는 컬렉션으로서 수정이 불가능합니다. 예를 들어, `5 to: 16 by: 2` 는 5, 7, 9, 11, 13 과 15라는 구성요소를 포함하고 있는 Interval이 됩니다. 이런 것들 (Interval 또는 Symbols) 은 at: 을 사용하면 색인이 가능하지만, at:put: 을 사용해서 값을 변경할 수는 없습니다.

성장가능 (Growable) Interval이나 Array의 인스턴스는 항상 고정된 크기를 가지게 됩니다. 다른 종류의 컬렉션 (sorted collection, ordered collection, 그리고 linked list) 은 일단 생성된 이후에 크기가 커질 수 있습니다. OrderedCollection 클래스는 배열보다 범용적이며, OrderedCollection은 동적으로 크기가 커질 수 있고, `at:` 과 `at:put:`은 물론이고 더 쓸모있는 `addFirst:`와 `addLast:` 등의 메서드도 가지고 있습니다.

중복 허용 (Accepts duplicates) Set 컬렉션은 중복된 요소를 걸러내지만, Bag은 그런작업을 하지는 않습니다. Dictionary, Set 그리고 Bag은 요소의 중복을 판단할때 각 요소에서 제공되는 =(equal) 메서드를 사용합니다; 이런 클래스의 Identity 등과 같은 변종은 두개의 인자 (파라메터) 가 같은 객체인가를 테스트하는 ==(double equal) 을 사용하며, Pluggable 같은 변종은 컬렉션의 생성시에 주어진 임의의 동치관계 (메서드) 를 사용합니다.

Arrayed Implementation	Ordered Implementation	Hashed Implementation	Linked Implementation	Interval Implementation
Array String Symbol	OrderedCollection SortedCollection Text Heap	Set IdentitySet PluggableSet Bag IdentityBag Dictionary IdentityDictionary PluggableDictionary	LinkedList SkipList	Interval

그림 9.4: 실행 테크닉에 의해 범주화된 몇 가지 컬렉션 클래스들

혼합가능 (Heterogeneous) 대부분의 컬렉션들은 요소가 어떤 종류라 하더라도 수용이 가능합니다. 하지만, String, CharacterArray 또는 Symbol 은 `Chracter`만을 수용할 수 있습니다. Array는 여러가지 객체를 수용할 수 있지만, ByteArray는 오직 Bytes 만 수용하며, IntergerArray는 오직 Integer 를 수용하고, FloatArray는 오직 Floats 만을 수용합니다. LinkedList는 Link ▷ accessing protocol 을 준수하는 요소만 수용가능한 제약을 가집니다..

9.3 컬렉션의 구현

기능에 의한 분류만이 관심이 되는건 아닙니다; 어떻게 콜렉션 클래스를 구현하는가에 대해서도 고려해야 하죠. 그림 **??** 의 내용처럼 구현에 사용되는 주요기술은 5가지가 있습니다.

1. 배열은, 컬렉션 객체 자신 (index가능) 의 인스턴스 변수에 요소를 수용합니다; 결과적으로 배열의 사이즈는 고정되며, 그 대신에 배열은 한번의 메모리 할당만으로 생성이 가능합니다.

2. OrderedCollections 과 SortedCollections는 요소를 배열에 수용하며 인스턴스 변수로 해당되는 배열을 참조합니다. 이렇게 함으로서 내부 배열의 사이즈보다 컬렉션이 커지면 보다 큰 배열로 교체가 가능합니다.

3. Set 과 Dictionary 같은 종류들도 저장장소를 위한 부가적인 배열을 참조합니다만, 이 배열을 참조할때는 해시테이블로 사용합니다. Bag 은 내부의 요소를 Key 로 하고 발생회수를 값으로 하며 두가지를 한쌍으로 취급하는 Dictionary 를 보조로 사용합니다.

4. LinkedLists는 표준적 singly-linked 형식을 사용합니다.

5. Interval은 두개의 끝점 및 단계의 크기를 나타내는 3 개의 정수를 수용합니다.

이 외에도 Array의 변종인 "WEEK" 및 Set 과 다양한 종류의 dictionary 들이 있습니다. 이런 컬렉션들은 요소를 "약하게" 보관합니다. 예를들면 보관하는 요소들에 대한 가비지컬렉션을 별도로 막지 않는 정도가 되겠군요. 스퀵가상머신은 이런 클래스들을 이미 알고 있으며, 특별하게 취급합니다.

스몰토크컬렉션들에 대해 좀 더 자세히 알고싶다면 LaLonde와 Pugh의 훌륭한 책 [8]을 참고해주시기 바랍니다.

9.4 주요 클래스에 대한 예제

지금부터 일반적이거나 중료한 컬렉션 클래스를 이용한 간단한 예제를 제시할 것입니다. 컬렉션의 주요 프로토콜들은 -구성요소에 접근하기 위한 at:, at:put:이 있으며, -구성요소를 추가하거나 제거하기 위한 add:, remove:가 있고, -컬렉션에서 몇가지 정보를 얻기 위한 size, isEmpty, include:가 있으며, -컬렉션의 반복을 위한 do:, collect:, select:가 있습니다. 각 컬렉션은 이런 프로토콜들을 실행하거나 실행하지 않을 수 있으며, 실행하는 경우에는 컬렉션의 의미에 맞게 프로토콜은 해석됩니다. 특정 프로토콜 및 좀 더 고급의 프로토콜을 식별하고싶다면 클래스 자체를 탐색하는것이 좀더 좋은 방법입니다.

여기서는 가장 일반적인 컬렉션 클래스들인 OrderedCollection, Set, SortedCollection, Dictionary, Interval 과 Array 를 집중적으로 살펴보겠습니다.

일반 생성 프로토콜(Common creation protocol) 컬렉션의 인스턴스를 만드는 데에는 몇가지 방법이 있습니다. `new:` 와 `with:` 를 사용하는 것이 가장 일반적인 방법입니다. new: anInteger는, 크기가 anInteger이며 모든 요소가 nil 인 컬렉션을 만듭니다. with: anObject는 컬렉션을 만들고, 만들어진 컬렉션에 anObject 를 추가합니다. 다른 컬렉션들은 이런 동작을 다르게 인식할 것입니다.

사용자는 초기 요소를 가지는 콜렉션을 만들기 위해 `with:`, `with:with:` 등의 메서드를 사용할 수 있습니다. 최대 6개의 요소를 지정할 수 있습니다.

```
Array with: 1     ⟶     #(1)
Array with: 1 with: 2     ⟶     #(1 2)
Array with: 1 with: 2 with: 3     ⟶     #(1 2 3)
Array with: 1 with: 2 with: 3 with: 4     ⟶     #(1 2 3 4)
Array with: 1 with: 2 with: 3 with: 4 with: 5     ⟶     #(1 2 3
    4 5)
Array with: 1 with: 2 with: 3 with: 4 with: 5 with: 6     ⟶
    #(1 2 3 4 5 6)
```

또한 어느 콜렉션의 모든 요소를 다른 종류의 콜렉션에 모두 추가하려면 addAll: 을 사용할 수 있습니다.

```
(1 to: 5) asOrderedCollection addAll: '678'; yourself  ⟶  an
    OrderedCollection(1 2 3 4 5 $6 $7 $8)
```

`addAll:`은 그 자체의 인수를 반환합니다만, 수신자를 반환하지는 않는다는 것에 주의해주세요.

또한 많은 컬렉션에서 `withAll:` 또는 `newFrom:`을 이용하여 만들수도 있습니다.

```
Array withAll: #(7 3 1 3)              ⟶  #(7 3 1 3)
OrderedCollection withAll: #(7 3 1 3)  ⟶  an
    OrderedCollection(7 3 1 3)
SortedCollection withAll: #(7 3 1 3)   ⟶  a SortedCollection(1
    3 3 7)
Set withAll: #(7 3 1 3)                ⟶  a Set(7 1 3)
Bag withAll: #(7 3 1 3)                ⟶  a Bag(7 1 3 3)
Dictionary withAll: #(7 3 1 3)         ⟶  a Dictionary(1-->7
    2-->3 3-->1 4-->3 )
```

```
Array newFrom: #(7 3 1 3)              ⟶  #(7 3 1 3)
OrderedCollection newFrom: #(7 3 1 3)  ⟶  an
    OrderedCollection(7 3 1 3)
SortedCollection newFrom: #(7 3 1 3)   ⟶  a SortedCollection(1
    3 3 7)
Set newFrom: #(7 3 1 3)                ⟶  a Set(7 1 3)
Bag newFrom: #(7 3 1 3)                ⟶  a Bag(7 1 3 3)
Dictionary newFrom: {1 --> 7. 2 --> 3. 3 --> 1. 4 --> 3}  ⟶  a
    Dictionary(1-->7 2-->3 3-->1 4-->3 )
```

이 두 개의 메서드들(`withAll`과 `newFrom:`)은 같지 않다는 것에 주의해 주세요.

`Dictionary class>>newFrom:` 은 인수로 관련된(비슷한) 컬렉션을 요구하지만, 이와는 다르게 `Dictionary class>>withAll:` 은 인수를 값(value)의 컬렉션으로 해석합니다.

배열

Array(배열)는, 고정 크기 컬렉션으로서 정수의 첨자(Integer indices)로 접근할 수 있습니다. C 언어의 관례와는 반대로, 스몰토크배열의 첫 번째 구성요소는 1 번 위치에 있으며 0 번 위치에서 시작하지 않습니다. 메서드 `at:` 과 `at:put:` 은 배열의 구성요소에 접근하기 위한 메인 프로토콜 입니다. at: anInteger는 anInteger의 내용이 의미하는 index 내 위치의 요소를 반환합니다. at: anInteger put: anObject는 anObject 를 anInteger에 해당하는 index에 집어넣습니다. 배열은 고정된 크기의 컬렉션이므로, 사용자는 배열의 끝부분에서 추가나 제거를 진행할 수 없습니다. 다음 코드는 5 의 크기를 가지는 배열을 만들어, 처음부터 세번째까지 3 개의 요소에 값들을 집어넣고 index의 첫번째 요소(index 1)를 반환합니다.

```
anArray := Array new: 5.
anArray at: 1 put: 4.
anArray at: 2 put: 3/2.
anArray at: 3 put: 'ssss'.
anArray at: 1    ⟶    4
```

Array 클래스의 인스턴스를 만드는데에는 몇가지 방법이 있습니다. 인스턴스를 만드는 작업을 위해서 `new:`, `with:`, 그리고 `#( )` 구문과 `{ }` 구문등을 사용할 수 있습니다.

`new:` 를 이용한 배열의 생성. `new: anInteger`는 `anInteger` 크기의 배열을 만듭니다. `Array new: 5`는 크기 5 의 배열을 만듭니다.

`with:` 를 이용한 배열의 생성. `with:` 메서드는 배열내 요소들의 값을 인수로 사용할 수 있습니다. 아래의 코드는 숫자 `4`, 분수 `3/2` 그리고 문자열 `'lulu'` 로 구성된 세 개의 요소를 가지는 배열을 생성합니다.

```
Array with: 4 with: 3/2 with: 'lulu'     ⟶     {4 . (3/2) .
    'lulu'}
```

`#()`로 리터럴 생성. `#()` 문법은 정적*static* 인 (또는 "literal") 요소를 가지는 리터럴 배열을 생성하는데, 이 정적*static* 이라고 하는 특성은 요소의 값이 실행 시점이 아닌 컴파일되었을때 이미 값을 가지고있어야 한다는 의미를 가집니다. 아래의 코드에서, 첫 번째 구성요소가 (literal) 숫자 1 이고 두 번째 요소는 (literal) 문자열 `'here'` 가 되는 크기 2 의 배열을 생성합니다.

```
#(1 'here') size     ⟶     2
```

이제, 만약 `#(1+2)`를 처리하셨다면, 3이라는 숫자값을 요소로 가지는 배열을 얻는것이 아니라, 대신 3 개의 구성요소 : 숫자 1, 심볼 `#+` 그리고 숫자2 의 3 개 요소를 가지는 `#(1 #+ 2)` 라는 배열을 얻게됩니다.

```
#(1+2)     ⟶     #(1 #+ 2)
```

이런 결과가 발생하는 이유는 `#()` 구문이 컴파일러로 하여금, 배열에 포함된 표현식을 그대로 (literal로) 해석하도록 지시하기 때문입니다. 인수로 주어지는 표현식은 조사되며 결과적으로 요소들은 새로운 배열로 수용됩니다. 리터럴 배열은 숫자, nil, true, false, symbol 그리고 문자열*string* 을 포함합니다.

{ }로 동적 배열 만들기. 마지막으로 알아볼 문법을 이용해서 동적배열을 만들 수 있습니다. `{ a . b }`는 : Array with: a with: b와 같은 의미가 됩니다. 그리고 특수한경우 { 기호와 } 기호로 둘러쌓인 표현식을 실행한다는 의미도 됩니다.

```
{ 1 + 2 }      ⟶      #(3)
{(1/2) asFloat} at: 1      ⟶      0.5
{10 atRandom . 1/3} at: 2      ⟶      (1/3)
```

요소 접근(Element Access). 모든 순차가능 컬렉션들의 요소들은 `at:`과 `at:put:`으로 접근이 가능합니다.

```
anArray := #(1 2 3 4 5 6) copy.
anArray at: 3      ⟶      3
anArray at: 3 put: 33.
anArray at: 3      ⟶      33
```

리터럴 배열을 수정하는 코드를 다룰 때 주의하십시오. 컴파일러는 리터럴 배열에 대해 공간할당을 단 한번만 진행합니다. 사용자가 배열을 복사하지 않는 한, 두 번째 코드를 평가하면 "리터럴" 배열은 예상하는 값을 가지고 있지 않을 수 있습니다. (객체복사를 하지 않았다면, 두 번째 단계에서, 리터럴 `#(1 2 3 4 5 6)`은 실제로 `#(1 2 33 4 5 6)`의 값을 가지게 됩니다!) 동적 배열들은 이런 문제를 갖고 있지 않습니다.

OrderedCollection

OrderedCollection은 크기변경이 가능한 컬렉션들 중의 하나이며, 이 컬렉션에서 요소는 순차적으로 추가할 수 있습니다. 이 컬렉션은 add:, addFirst:. addList:, 그리고 addAll: 등과 같은 다양한 메서드를 제공합니다:

```
ordCol := OrderedCollection new.
ordCol add: 'Seaside'; add: 'SqueakSource'; addFirst:
    'Monticello'.
ordCol      ⟶      an OrderedCollection('Monticello' 'Seaside'
    'SqueakSource')
```

요소의 제거 (Removing Elements). 메서드 remove: anObject는 컬렉션내의 요소중 anObject와 일치하는걸 찾은후 첫번째 결과에 해난하는 요소를 제거합니다. 만약 컬렉션에 이러한 객체 (anObject) 가 포함되어 있지 않다면, 에러를 발생시킵니다.

```
ordCol add: 'Monticello'.
ordCol remove: 'Monticello'.
ordCol      ⟶      an OrderedCollection('Seaside' 'SqueakSource'
    'Monticello')
```

`remove:ifAbsent:`라는 remove:의 변형메서드가 있는데, 이 `remove:ifAbsent:` 메서드의 두번째 인수에는 삭제하려고 시도한 요소가 콜렉션에 없는경우의 동작을 블록으로 넣을 수 있습니다.

```
res := ordCol remove: 'zork' ifAbsent: [33].
res      ⟶      33
```

변환 (Conversion) asOrderedCollection이라는 메시지를 을 발송함으로써, 배열 또는 다른 컬렉션으로부터 OrderedCollection 을 얻어내는 것이 가능합니다.

```
#(1 2 3) asOrderedCollection      ⟶      an OrderedCollection(1 2
    3)
'hello' asOrderedCollection      ⟶      an OrderedCollection($h
    $e $l $l $o)
```

Interval

Interval이라는 클래스는 숫자의 범위를 제공합니다. 예를 들어,, 숫자 1 에서 100 까지의 인터벌은 다음과 같이 정의됩니다.

```
Interval from: 1 to: 100      ⟶      (1 to: 100)
```

이 인터벌 객체의 printString 결과를 보면, Number 클래스에 to: 라고 하는 편리한 메서드가 있고 이것을 사용해서 Interval 객체를 생성할 수 있다는걸 알 수 있습니다.

```
(Interval from: 1 to: 100) = (1 to: 100)      ⟶      true
```

다음과 같이 2 개의 숫자들 사이의 단계를 지정하기 위해 Interval `Interval class>>from:to:by:` 또는 `Number>>to:by:` 를 사용하는것도 가능합니다.

```
(Interval from: 1 to: 100 by: 0.5) size     ⟶     199
(1 to: 100 by: 0.5) at: 198     ⟶     99.5
(1/2 to: 54/7 by: 1/3) last     ⟶     (15/2)
```

Dictionary

Dictionary는 키(keys)를 사용해서 요소에 접근하는 중요한 컬렉션 입니다. Dictionary에서 가장 일반적으로 사용되는 메시지는 at:, at:put:, at:ifAbsent:, 그리고 keys와 values가 있습니다.

```
colors := Dictionary new.
colors at: #yellow put: Color yellow.
colors at: #blue put: Color blue.
colors at: #red put: Color red.
colors at: #yellow     ⟶     Color yellow
colors keys     ⟶     a Set(#blue #yellow #red)
colors values     ⟶     {Color blue . Color yellow . Color red}
```

Dictionary는 키를 비교하는것으로 동일성을 검사합니다. 만약 = 를 사용하여 비교할 때, true가 리턴되면 2 개의 키는 동일한 것으로 간주합니다. 찾기 힘든 버그중 제일 많이 일어나는 상황은, Key 로서 사용되는 객체에 = 메서드를 재정의했지만, hash 메서드를 재정의하지 않는 경우입니다.

Dictionary 클래스는 콜렉션 계층이 서브클래스*subclass* 에 기초하며 서브타이핑*subtyping* 은 아니라는것을 명확하게 보여줍니다. 심지어 Dictionary가 Set의 서브클래스 입니다만, 일반적으로 Dictionary 를 Set와 같이 사용하려고 하지는 않습니다. Dictionary는 -> 메시지를 이용해서 만들어진 key-value 연관성*associations* 의 세트로 구성된것이 분명합니다. 그렇기때문에 Dictionary는 연관성을 가지는 콜렉션으로부터 만들어질 수도 있으며, Dictionary 를 연관성을 가지는 배열로 변환할 수고 있습니다.

```
colors := Dictionary newFrom: { #blue ⟶ Color blue.
    #red ⟶ Color red. #yellow ⟶ Color yellow }.
colors removeKey: #blue.
colors associations     ⟶     {#yellow ⟶ Color yellow .
    #red ⟶ Color red}
```

IdentityDictionary. Dictionary에서 2 개의 키가 동일한지를 판단하기 위해 메시지 = 와 hash의 결과를 사용하는 반면에, 클래스 IdentityDictionary는 이런 값들 대신에, 키의 identity(메시지==)를 사용합니다. IdentityDictionary는 2 개의 키가 동일한 객체일 경우만, 동일한 것으로 판단합니다.

Symbol들은 자주 Key 로서 사용됩니다만, 이런경우 IdentityDictionary를 사용하는 것은 자연스러운 일이며, 왜냐하면 Symbol은 global에서 고유하게 보증되기 때문입니다. 한편, Key가 String 인 경우라면, Dictionary 클래스를 그대로 사용하는 것이 좋습니다. 그렇게 하지 않는다면 작업에서 문제가 생기겠지요:

```
a := 'foobar'.
b := a copy.
trouble := IdentityDictionary new.
trouble at: a put: 'a'; at: b put: 'b'.
trouble at: a     ⟶     'a'
trouble at: b     ⟶     'b'
trouble at: 'foobar'     ⟶     'a'
```

위의 예에서 a와 b는 다른 객체이기 때문에, 다른 객체로서 취급됩니다. 흥미로운 부분이 있는데, 리터럴 `'foobar'` 는 단지 한번만 할당되므로, a와 실제로 동일한 객체가 됩니다. 대부분 이런 명확하지 않는 동작에 의존하는 코드를 만들려고 싶어하지는 않겠죠! 일반적으로 Dictionary는 'foobar' 와 같은 Key에 대해서는 동일한 값을 반환합니다.

IdentityDictionary Key 로서는 global 하게 고유한 객채 (Symbols 또는 SmallIntegers 등) 만을 사용해주시길 바라며, String (또는 다른 객체) 은 일반적으로 Dictionary의 Key 로서 사용해주세요..

global 변수인 `Smalltalk` 는 `IdentityDictionary` 의 서브클래스인, `System Dictionary` 의 인스턴스인것에 주의해주시길 바라며, 이런 이유때문에 모든 Key는 Symbol(실제로는 8 비트 문자밖에 가질 수 없는 ByteSymbol 입니다) 이 됩니다.

```
Smalltalk keys collect: [ :each | each class ]  ⟶  a
    Set(ByteSymbol)
```

keys 또는 values 메시지를 Dictionary에 보내면 결과는 Set가 됩니다만, Set 클래스에 대해서는 이제부터 다시 설명하도록 하겠습니다.

Set

Set 클래스는, 수학에서 말하는 집합처럼 작동하는 컬렉션이며, 중복요소를 가지지 않고, 요소간에 순서를 가지지 않습니다. Set에 요소를 추가하려면 add: 메시지를 이용하지만, at: 메시지를 이용해서 요소에 접근하는건 불가능합니다. Set에 집어넣어진 객체들은, 메서드 `hash`와 메서드 `=` 을 가지고 있어야 합니다.

```
s := Set new.
s add: 4/2; add: 4; add:2.
s size     ⟶     2
```

Set 을 만들때에는 Set `Set class>>newFrom:` : 또는 변환 메시지 `Collection >>asSet` 를 사용하면 됩니다.

```
(Set newFrom: #( 1 2 3 1 4 )) = #(1 2 3 4 3 2 1) asSet     ⟶
     true
```

컬렉션으로부터 중복된 사본들을 제거하는 편리한 방법으로서 asSet 을 사용할 수 있습니다.

```
{ Color black. Color white. (Color red + Color blue + Color
    green) } asSet size     ⟶     2
```

red + blue + green = white가 되는것에 주의해 주세요.

Bag은 중복된 것들을 허용한다는 것을 제외하면 Set와 매우 비슷합니다:

```
{ Color black. Color white. (Color red + Color blue + Color
    green) } asBag size     ⟶     3
```

Set 연산들인 union, intersection 그리고 membership test는 컬렉션 메시지 union:, intersection: 그리고 includes:. 에 의해 실행됩니다. 수신자는 Set로 먼저 변환되므로, 이 연산들은 모든 종류의 컬렉션들을 위해 작동됩니다!

```
(1 to: 6) union: (4 to: 10)      ⟶     a Set(1 2 3 4 5 6 7 8 9
    10)
'hello' intersection: 'there'      ⟶     'he'
#Smalltalk includes: $k            ⟶     true
```

이후에 설명하겠습니다만, Set의 각 요소는 iterators(9.5절 참조) 를 사용해서 접근할 수 있습니다.

SortedCollection

`OrderedCollection` 과는 반대로, `SortedCollection` 은 보유하고있는 요소들의 정렬순서를 유지합니다. 기본적으로 SortedCollection은 정렬 순서를 정하는 작업에 `=` 메시지를 사용하므로, 비교 객체들 (`<`, `=`, `>`, `>=`, `between:and:`) 의 프로토콜을 정의하는 (8장 을 보십시오) 추상 클래스 Magnitude의 서브클래스의 인스턴스들을 정렬할 수 있습니다.

SortedCollection 을 만들기 위해서는, 새로운 인스턴스를 만들고 요소들을 그 인스턴스에 추가하면 됩니다.

```
SortedCollection new add: 5; add: 2; add: 50; add: --10;
    yourself.  ⟶  aSortedCollection(--10 2 5 50)
```

좀 더 일반적인 경우를 보자면, 사용자가 asSortedCollection이라는 변환메시지를 현존하는 컬렉션에 전송하는 방법도 있습니다.

```
  #(5 2 50 ----10) asSortedCollection  ⟶  a
    SortedCollection(--10 2 5 50)
```

다음 FAQ(질답) 에 대한 답변으로는 아래의 예제가 답이 될거같군요 :

> FAQ: collection의 내부를 어떻게 정렬하나요?
> ANSWER: `asSortedCollection` 메시지를 전송하면 됩니다.

```
  'hello' asSortedCollection  ⟶  a SortedCollection($e $h $l $l
    $o)
```

위의 예제로부터 정렬을 진행한 결과를 String 으로 되돌리기 위해서는 어떻게 하면 좋을까요? 불행하게도 asString이 반환하는 printString은 우리 예상대로 되지는 않습니다.

```
  hello' asSortedCollection asString  ⟶  'a SortedCollection($e
    $h $l $l $o)'
```

위의 내용에 대한 정확한 답은 `String class>>newFrom:`, `String class>>withAll:` 또는 `Object>>as:` 등을 사용하는게 되겠습니다.

```
'hello' asSortedCollection as: String          ⟶     'ehllo'
String newFrom: ('hello' asSortedCollection)   ⟶     'ehllo'
String withAll: ('hello' asSortedCollection)   ⟶     'ehllo'
```

모든 요소가 서로 비교가 가능하다는 전제하에, SortedCollection은 다른종류의 요소를 가지는게 가능합니다. 예를들어 정수 (integers), 부동소수점수 (floats), 분수 (fractions) 와 같은 비교가능한 다양한 종류의 숫자들을 요소로 가질 수 있다는 의미입니다.

```
{ 5. 2/--3. 5.21 } asSortedCollection  ⟶  a
    SortedCollection((--2/3) 5 5.21)
```

메서드 <= 를 정의하지 않고있는 객체들을 정렬하려하거나, <= 이외의 다른 기준으로 정렬을 하려고 한다고 생각해 보도록 하겠습니다. 사용자는 이 경우 sortblock이라고 불리는 2 개로 내부가 나뉜 상태로 구성된 블록을 전달하는 것으로 정렬은 가능해집니다. 예를들면 Color 클래스는 Magnitude가 아니고 <= 메서드를 가지고 있습니다만, 아래처럼 sortblock 을 지정하는것으로, 색에 대한 값을 명도 (밝기에 대한 기준) 에 의해서 정렬할 수 있습니다.

```
col := SortedCollection sortBlock: [:c1 :c2 | c1 luminance <= c2
    luminance].
col addAll: { Color red. Color yellow. Color white. Color black }.
col     ⟶     a SortedCollection(Color black Color red Color
    yellow Color white)
```

String

스몰토크에서 문자열 (`String`) 은 Chracter의 컬렉션으로 표현됩니다. 이렇게 되어있는 컬렉션은 순차가능*sequenceable*, 색인가능*indexable*, 수정가능*mutable* 그

리고 혼합가능*homogeneous* 하며 같은종류의 요소, 즉 Chracter의 인스턴스만 가질 수 있습니다. String은 일반적으로 String 리터럴을 '(single quote)로 둘러싸는것으로 생성하며, Array 처럼 String은 전용 문법을 가지고 있습니다만, 일반적인 컬렉션 메서드도 잘 작동합니다.

```
'Hello'     ⟶     'Hello'
String with: $A     ⟶     'A'
String with: $h with: $i with: $!     ⟶     'hi!'
String newFrom: #($h $e $l $l $o)     ⟶     'hello'
```

사실, String은 추상클래스입니다. String 을 인스턴스화 하는경우 실제로 얻는것은 8-bit ByteString 또는 32-bit WideString 입니다. 설명을 간단하게 하기위해서, 이런 차이는 무시하고 String의 인스턴스에 관해서만 알아보도록 하겠습니다.

String 형태의 2 개의 인스턴스는 ,(comma-쉼표)로 연결될 수 있습니다.

```
s := 'no', ' ', 'worries'.
s     ⟶     'no worries'
```

String은 수정가능*mutable* 한 컬렉션이기 때문에, 사용자는 메서드 at:put: 을 사용하여 내용을 변경할 수 있습니다.

```
s at: 4 put: $h; at: 5 put: $u.
s     ⟶     'no hurries'
```

쉼표 메서드는 Collection에서 정의되어 있기 때문에, 모든 종류의 컬렉션에 대해서 동작이 가능하다는 것을 주의해 주십시오!

```
(1 to: 3) , '45'     ⟶     #(1 2 3 $4 $5)
```

아래의 예제에서 볼 수 있듯이, 이미 존재하는 문자열을 replaceAll:with: 또는 replaceFrom:to:with: 등을 사용해서 변경하는것도 가능합니다. 인수로 주어지는 문자수와 지정하는 간격의 길이가 같아야 하는것에(replaceFrom의 경우) 주의해주세요[3].

[3]일본어버전의 Pharo by example 을 보면 같지 않은경우 에러가 발생한다고 되어 있습니다.

```
s replaceAll: $n with: $N.
s     ⟶     'No hurries'
s replaceFrom: 4 to: 5 with: 'wo'.
s     ⟶     'No worries'
```

위에서 언급된 메서드들과는 다르게도, copyReplaceAll: 메서드는 새로운 문자열을 생성합니다.(신기하게도 이 메서드의 인수는 개별의 문자보다는 부분문자열로 취급되기 때문에 크기가 같아야할 필요는 없습니다)

```
s copyReplaceAll: 'rries' with: 'mbats'  ⟶  'No wombats'
```

이 메서드의 구현내용을 살펴보면, 사실 String 전용으로 정의된것이 아니고 어떤 종류의 SequenceableCollection에서도 사용이 가능하다는것을 알 수 있습니다. 따라서 아래의 내용도 동작하게 되는거죠.

```
(1 to: 6) copyReplaceAll: (3 to: 5) with: { 'three'. 'etc.' }
     ⟶  #(1 2 'three' 'etc.' 6)
```

String Matching. 메시지 `match:` 를 발송함으로써 패턴이 String 과 매치되는지의 여부를 요청하는 작업이 가능합니다. 패턴에서 *(asterisk) 기호는 임의의 길이를 의미하며, #(sharp) 기호는 임의의 1 개문자를 의미합니다. `match:`는 패턴에서 문자열을 받을는 경우로 사용하며 문자열에 패턴을 보내는 상황에 사용되는건 아니라는걸 주의해 주세요.

```
'Linux *' match: 'Linux mag'     ⟶     true
'GNU/Linux #ag' match: 'GNU/Linux tag'     ⟶     true
```

다른 유용한 메서드로는 findString: 입니다.

```
'GNU/Linux mag' findString: 'Linux'
          ⟶  5
'GNU/Linux mag' findString: 'linux' startingAt: 1 caseSensitive:
    false  ⟶  5
```

Perl 수준의 좀더 진보된 패턴 매칭 도구도 사용이 가능합니다만, 표준 이미지에 포함되어 있지는 않습니다.[4]

[4] 우리는 Vassili Bykov의 정규 표현식 패키지를 강력하게 추천합니다. 이 패키지는 http://www.squeaksource.com/Regex.html에서 찾아보실 수 있습니다.

String에 대한 테스트. 아래의 예제들은 String 뿐만 아니라 다른 일반적인 컬렉션들에서도 정의되고있는 isEmpty, includes:와 anySatisfy: 등의 메시지를 string에 사용했을때의 동작을 보여줍니다.

```
'Hello' isEmpty     ⟶     false
'Hello' includes: $a     ⟶     false
'JOE' anySatisfy: [:c | c isLowercase]     ⟶     false
'Joe' anySatisfy: [:c | c isLowercase]     ⟶     true
```

String templating. String templating 을 관리하는데 유용한 `format:`, `expandMacros` 그리고 `expandMacrosWith:` 와 같은 3 개의 메시지들이 있습니다.

```
'{1} is {2}' format: {'Squeak' . 'cool'}     ⟶     'Squeak is
    cool'
```

`expandMacros` 계열의 메시지들은 캐리지 리턴을 위해 `<n>` 을 사용한 변수 교체를 제공하며, 도표작성을 위해 `<t>` 를 반환하고 인수들을(`<1p>`, `<2p>`, 작은 따옴표로 문자열을 둘러싸서) 위해 `<1s>`, `<2s>`, `<3s>`를 반환하고, 조건문을 위해 `<1?value1:value2>`를 반환하는것등을 지원합니다.

```
'look--<t>--here' expandMacros                                  ⟶
    'look-- --here'
'<1s> is <2s>' expandMacrosWith: 'Squeak' with: 'cool'  ⟶
    'Squeak is cool'
'<2s> is <1s>' expandMacrosWith: 'Squeak' with: 'cool'  ⟶
    'cool is Squeak'
'<1p> or <1s>' expandMacrosWith: 'Squeak' with: 'cool'  ⟶
    \emph{'Squeak} or Squeak '
'<1?Quentin:Thibaut> plays' expandMacrosWith: true      ⟶
    'Quentin plays'
'<1?Quentin:Thibaut> plays' expandMacrosWith: false     ⟶
    'Thibaut plays'
```

그외 유틸리티 메서드 String 클래스는 메시지 `asLowercase`, `asUppercase` 와 `capitalized` 를 포함한 여러 개의 다른 유틸리티를 제공합니다.

```
'XYZ' asLowercase     ⟶     'xyz'
'xyz' asUppercase     ⟶     'XYZ'
'hilaire' capitalized     ⟶     'Hilaire'
```

```
'1.54' asNumber      ⟶      1.54
'this sentence is without a doubt far too long' contractTo: 20
      ⟶      'this sent\ldotstoolong'
```

printString 메시지를 발송하여 객체에게 보유한 문자열 표현을 요청하는 작업과, asString 메시지를 이용해서 문자열로 변환하는 작업에는 일반적으로 차이점이 있다는 것에 주의하십시오. 여기에 그 차이에 대한 예제가 있습니다.

```
#ASymbol printString      ⟶      '#ASymbol'
#ASymbol asString      ⟶      'ASymbol'
```

Symbol은 string 과 비슷하지만, global에서 고유성 유지가 보장된다는것이 차이가 있습니다. 이러한 이유 때문에 Symbol들은 특별히 IdentityDictionary의 인스턴스인 경우, Dictionary 들을 위한 key 들로서 string 보다는 Symbol 을 더 선호하는 편입니다. string 과 symbol 에 관해 더 많은 내용을 원하신다면 8장 을 참고해주세요.

9.5 컬렉션의 반복자 (Collection iterators)

스몰토크에서 반복문이나 조건문은, 컬렉션, 정수, 블록과같은 객체에 대한 단순한 메시지 전송입니다 (3장 을 참고해주세요). 스몰토크의 콜렉션계층은, 초기값부터 마지막값까지 변화하는 수치를 인자로 블록을 평가하는 to:do: 같은 저수준 메시지를 포함한 여러가지 고수준의 반복자*iterator*[5]를 제공합니다. 이런 반복자의 사용은, 프로그램의 코드를 좀 더 튼튼하고 간결하게 만들어 줍니다.

Iterating (do:)

do: 메서드는 기본적인 컬렉션 반복자 메서드입니다. do: 메서드는 그 자체의 인자 (단일 인자를 취하는 블록) 를 수신자의 각 요소에 순서대로 적용합니다.

[5] Iterator는 한국어로 대부분 반복자라고 부릅니다만, wikipedia의 경우에는 객체지향 프로그래밍에서 container에 제공되는것이라고 언급합니다. 이런 컨테이너는 사실 smalltalk에서는 컬렉션이라고 할만하죠. 여기서는 일반적으로 한국에서 사용하듯이 반복자라는 용어를 사용하도록 하겠습니다.

다음 예제는 수신자에 들어 있는 모든 문자열을 Transcript에 print 합니다.

```
#('bob' 'joe' 'toto') do: [:each | Transcript show: each; cr].
```

변종(Variants) do: 메서드에는 많은 변종이 있는데, 예를들면 do:without:, doWithIndex:, reverseDo: 등이 해당되겠죠. indexed 컬렉션(Array, OrderedCollection, SortedCollectino) 을 위한 `doWithIndex:` 메서드는 현재 index에 대한 접근등을 제공합니다. 이 doWithIndex: 메서드는 Number 클래스에서 정의된 to:do:와 관련이 있습니다.

```
#('bob' 'joe' 'toto') doWithIndex: [:each :i | (each = 'joe')
    ifTrue: [ ↑ i ] ]     ⟶     2
```

orderd collection의 reverseDo:는 컬렉션의 정렬을 역순으로 처리합니다.

아래의 코드에서 do:separatedBy: 라는 재미있는 메서드를 볼 수 있습니다. 이 메시지는 두개의 요소 사이를 대상으로 두번째 인자로 주어진 블록을 실행합니다.

```
res := ''.
#('bob' 'joe' 'toto') do: [:e | res := res, e ] separatedBy: [res
    := res, '.'].
res     ⟶     'bob.joe.toto'
```

위의 방법은 효율적이라고 볼 수는 없으며, 결과를 buffer에 쓰기 위해서는 stream 을 사용하는게 더 좋은 방법이라는걸 주의해주세요(10장 을 참고).

```
String streamContents: [:stream | #('bob' 'joe' 'toto')
    asStringOn: stream delimiter: '.' ]     ⟶     'bob.joe.toto'
```

Dictionaries. 메시지 do:가 dictionary 로 전송될때 검사되는 요소는 value 가 되며 association[연관] 은 되지 않습니다. 이런 경우 keysDo:, valuesDo: associationsDo: 등을 사용하는것이 더 적당하며, 각 메서드들은 Keys, values 또는 associations에 대해 반복 실행됩니다.

```
colors := Dictionary newFrom: { #yellow --> Color yellow. #blue --
    > Color blue. #red --> Color red }.
```

```
colors keysDo: [:key | Transcript show: key; cr].
                        `displays the
    keys'
colors valuesDo: [:value | Transcript show: value;cr].
                        `displays the
    values'
colors associationsDo: [:value | Transcript show: value;cr].
                        `displays the
    associations'
```

Collecting results (collect:)

콜렉션의 요소를 처리하고, 그 결과를 새로운 컬렉션으로 만들기를 원한다면, do: 를 사용하는것보다 collect: 또는 그 외의 반복자 메서드를 이용하는 것이 보다 좋은 방법이 될겁니다. 이런 메서드의 대부분은, Collection이나 서브클래스에서 열거*enumerating* 프로토콜쪽에서 찾을 수 있습니다.

예를들어 각 요소의 2 배로 하는 새로운 컬렉션을 만든다고 생각해 보겠습니다. do: 를 사용한다면 다음과 같은 코드가 될겁니다.

```
double := OrderedCollection new.
#(1 2 3 4 5 6) do: [:e | double add: 2 * e].
double      ⟶      an OrderedCollection(2 4 6 8 10 12)
```

collect: 메서드는, 인자로 건네받은 블록을 각 요소에 대해 실행하고, 결과를 수용하는 새로운 콜렉션을 반환합니다. do: 대신에 collect: 를 사용하면 코드는 매우 간결해지게 됩니다:

```
#(1 2 3 4 5 6) collect: [:e | 2 * e]      ⟶      #(2 4 6 8 10 12)
```

`do:` 와 비교되는 `collect:`의 장점은 아래의 예제에서 더 극적으로 확인되는데, 예제는 정수의 콜렉션을 준비하고, 그 절대값을 결과로 받는경우가 되겠습니다.

```
aCol := #( 2 --3 4 --35 4 --11).
result := aCol species new: aCol size.
1 to: aCol size do: [ :each | result at: each put: (aCol at:
    each) abs].
result      ⟶      #(2 3 4 35 4 11)
```

위의 표현식과 대비되는 아래의 간결한 표현식을 비교해 보시기 바랍니다.

```
#( 2 --3 4 --35 4 --11) collect: [:each | each abs ]     ⟶
    #(2 3 4 35 4 11)
```

두 번째 방법이 더 좋은 이유는, 이런 방법이 Set이나 Bag에서도 동일하게 작동할거라는 점입니다.

일반적으로 컬렉션의 각 요소들에 메시지를 발송하는것을 원하지 않는다면, do: 사용을 피하는 것이 바람직합니다.

메시지 collect: 를 송신하면 수신자와 동일한 종류의 컬렉션을 반환한다는 사실에 주의해주세요. 이러한 이유 때문에, 다음 코드는 실패하게 됩니다. (String은 Integer 값들을 가지고 있지 않습니다.)

```
'abc' collect: [:ea | ea asciiValue ] `error!'
```

그대신 String 을 Array 또는 OrderedCollection 으로 변환해서 처리할 수 있습니다.

```
'abc' asArray collect: [:ea | ea asciiValue ]     ⟶     #(97 98
    99)
```

실제로 `collect:`는 수신자와 정확히 동일한 클래스에 대한 컬렉션 반환을 보장하지 않으며 오직 동일한 "종(species)" 만을 반환합니다. Interval의 경우 유사한 종류는 사실 Array 입니다.

```
(1 to: 5) collect: [ :ea | ea * 2 ]     ⟶     #(2 4 6 8 10)
```

요소들을 선택하고 거부하기

select: 메서드는 특별한 조건을 만족시키는 수신자의 요소들을 반환합니다:

```
(2 to: 20) select: [:each | each isPrime]     ⟶     #(2 3 5 7
    11 13 17 19)
```

reject: 메서드는 select:와는 반대되는 작업을 수행합니다:

```
(2 to: 20) reject: [:each | each isPrime]     ⟶     #(4 6 8 9
    10 12 14 15 16 18 20)
```

detect: 로 요소를 식별하기

detect: 메서드는 수신자의 요소중에서 주어진 블록 인자를 만족하는 첫번째 요소를 반환합니다.

```
'through' detect: [:each | each isVowel]     ⟶     $o
```

메서드 detect:ifNone은 메서드 detect:의 변종입니다. 이 메서드의 두 번째 인자(블록)은, 첫번째 블록에서 일치되는 요소가 없을 때, 실행됩니다.

```
Smalltalk allClasses detect: [:each | '*java*' match: each
    asString] ifNone: [ nil ]     ⟶     nil
```

inject:into:로 결과들을 모으기

함수형 프로그래밍 언어들은, 자주 몇 가지 이항 연산자를 컬렉션의 모든 구성요소에 반복적으로 적용해서 결과를 받기 위해 fold 또는 reduce라고 지칭되는 좀 더 고수준의 정렬 기능을 제공합니다. 스퀵에서 이런 기능은 Collection»>inject:into: 를 사용하면 됩니다.

이 메서드의 첫 번째 인자는 초기 값이며, 두 번째 인자는 지금까지의 결과와 각각의 구성 요소에 차례대로 적용된 2 개의 인자 블록이 됩니다.

inject:into:의 간단한 예로서 숫자들의 컬렉션에 대한 총합을 계산해 보겠습니다. 가우스(Gauss) 처럼, 스퀵에서는 1 에서 부터 시작한 100 까지의 정수*Integer* 들의 총합을 구하기 위해 아래와 같은 표현식을 작성하면 됩니다:

```
(1 to: 100) inject: 0 into: [:sum :each | sum + each ]     ⟶
     5050
```

또다른 예로서, 분수들을 계산할때 1 개의 인자 블록을 쓴다면 다음과 같이 작성할 수 있겠죠:

```
factorial := [:n | (1 to: n) inject: 1 into: [:product :each |
    product * each ] ].
factorial value: 10     ⟶     3628800
```

그 외의 메시지들

count: 메시지 count:는 조건식을 만족시키는 요소의 개수를 반환합니다. 조건식은 Boolean 블록으로 작성합니다.

```
Smalltalk allClasses count: [:each | 'Collection*' match: each
    asString ]     ⟶     2
```

includes: 메시지 includes:는 인자가 컬렉션에 포함되었는지의 여부를 점검합니다.

```
colors := {Color white . Color yellow. Color red . Color blue .
    Color orange}.
colors includes: Color blue.     ⟶     true
```

anySatisfy: 메시지 anySatisfy:는 만약 최소 1 개의 컬렉션 요소가 인자에 의해 제시된 조건식을 만족시킬 경우 true 로 답변합니다.

```
colors anySatisfy: [:c | c red > 0.5]     ⟶     true
```

9.6 컬렉션 사용에 대한 몇 가지 힌트

add: 메서드를 사용할 때 발생되는 일반적인 실수: 다음의 에러는 가장 많이 발생되는 스몰토크코딩에서의 실수 중 하나 입니다.

```
collection := OrderedCollection new add: 1; add: 2.
collection     ⟶     2
```

여기서 보게되는 변수 collection은 새롭게 만들어진 컬렉션이 아니라, 마지막에 추가된 숫자를 보관합니다. 왜냐하면 메서드 add:는 추가된 요소를 반환하되, 수신자는 반환하지 않기 때문입니다.

다음 코드에서 예상대로의 결과를 얻을 수 있습니다:

```
collection := OrderedCollection new.
collection add: 1; add: 2.
collection     ⟶     an OrderedCollection(1 2)
```

그외에 yourself 메시지를 사용할 수 있는데, 이 메시지는 cascade 된 메시지의 수신자를 반환합니다.

```
collection := OrderedCollection new add: 1; add: 2; yourself
      ⟶      an OrderedCollection(1 2)
```

반복을 수행하고 있는 콜렉션으로부터 요소를 삭제. 자주 일어날 수 있는 또 다른 실수는, 현재 반복작업을 수행하는 컬렉션으로부터 요소를 제거하는 것입니다. remove: 를 이용하는 경우에 말이죠.

```
range := (2 to: 20) asOrderedCollection.
range do: [:aNumber | aNumber isPrime ifFalse: [ range remove:
    aNumber ] ].
range      ⟶      an OrderedCollection(2 3 5 7 9 11 13 15 17 19)
```

9 와 15 가 걸러져야 했기 때문에, 이 결과는 잘못된 결과입니다.

해결 방법은, 검사를 하기 전에 컬렉션을 복사해 두는 것입니다.

```
range := (2 to: 20) asOrderedCollection.
range copy do: [:aNumber | aNumber isPrime ifFalse: [ range
    remove: aNumber ] ].
range      ⟶      an OrderedCollection(2 3 5 7 11 13 17 19)
```

= 와 hash는 한번에 양쪽 모두 재정의 되어야 한다. 에러중 발견하기 어려운 경우는, 사용자가 = 을 재정의하면서 hash는 재정의하지 않는 경우입니다. 이런경우 Set에 분명히 넣은 요소가 어느새 없어진다는 현상등이 나타납니다. Kent Beck이 제안한 해결책은 xor: 을 이용해서 bash를 재정의 하는 것입니다. 예를 들어, 만약 제목과 저자가 같은경우, 두 개의 책을 동일한 것으로 판단하고 싶을때, 다음과 같이 = 를 재정의할 뿐만 아니라 hash도 재정의해야 할 것입니다.

Method 9.1: = 과 *hash* 재정의 하기

```
Book>>= aBook
  self class = aBook class ifFalse: [↑ false].
  ↑ title = aBook title and: [ authors = aBook authors]

Book>>hash
  ↑ title hash xor: authors hash
```

만약 Set의 요소 또는 Dictionary에 대한 key 를, 그 자체의 hash 값의 변경이 가능한 객체처럼-수정가능한 객체를 사용하는 경우에는, 또 다른 끔찍한 문제가 발생합니다. 당신이 디버깅을 정말로 좋아하는게 아니라면, 이런 경우에는 수정가능한 객체를 사용하지 말아주세요!

9.7 9장 요약

스몰토크의 컬렉션 계층은, 다양한 종류의 컬렉션을 동일하게 조작하기 위한 공통의 어휘를 가지고 있습니다.

- 컬렉션에서 볼 수 있는 가장 큰 차이는, `SequenceableCollections`은 주어진 순서에 따라 요소들을 보관하는것에 비해, `Dictionary` 및 `Dictionay`의 서브클래스들은 Key-Value의 연관 쌍을 보관하며, `Sets`와 `Bags`는 다루는 요소의 순서에 대한 규칙을 가지지 않는다는 점입니다.
- 컬렉션에 메시지 `asArray`, `asOrderedCollection` 를 발송함으로써 대부분의 컬렉션들을 다른 종류의 컬렉션으로 변환할 수 있습니다.
- 컬렉션을 정렬하기 위해서는, 정렬을 원하는 컬렉션에 `asSortedCollection` 을 발송합니다.
- 리터럴의 `Array`는 `#( ... )` 같은 특별한 문법으로 만들 수 있습니다. 동적 Array는 `{ ... }` 문법으로 만들수 있습니다.
- `Dictionary`는 key 를 비교할때 동치성(equality)을 사용합니다. String의 인스턴스를 Key 로 사용할때 가장 유용합니다. 그대신 `IdentityDictionary`는 객체의 동일성(object identity)으로 key 를 판단합니다. 이것은 key 로 Symbol 을 시용하거나, 객체의 참조를 값에 대입할때 적합합니다.
- `String`은 일반적인 컬렉션 메시지의 처리가 가능합니다. 게다가, `String`은 Pattrern-Matching의 기본적인 형태도 지원합니다. 좀더 진보된 프로그램에는, 일반적인 matching 대신 RegEx package 를 사용하면 됩니다.

- 기본적인 반복자 메시지는 `do:` 입니다. 이 메시지는 컬렉션의 요소를 수정하거나, 각 요소에 같은 메시지를 발송하는 작업에 대한 코드등에 유용하게 사용될 수 있습니다.
- 사실 `do:` 메서드 대신, 동일한 방식의 처리를 위해 `collect:`, `select:`, `reject:`, `includes:`, `inject:into:` 및 그외의 고수준인 메시지를 사용하는 것이 좀 더 일반적입니다.
- 반복을 수행하고 있는 컬렉션의 요소를 절대로 제거하지 마십시오. 만약 컬렉션의 요소를 수정하셔야 한다면, 요소를 다른 객체로 복사한다음 수정해야 합니다.
- 만약 `=` 메서드를 재지정해야 한다면 `hash` 메서드 또한 재지정해야 한다는걸 기억하시기 바랍니다.

제 10 장

Streams

Stream은 순차적 컬렉션, 파일, 네트워크 스트림같은 순차적 요소를 반복처리하는데 유용합니다. Stream은 경우에 따라 읽기 또는 쓰기가 가능하거나 두가지 다 가능합니다. 읽기와 쓰기작업은 Stream 내의 현재위치를 기준으로 진행됩니다. Stream은 쉽게 컬렉션으로 변환이 가능하며 그 반대의 변환도 가능합니다.

10.1 두 요소에 대한 순서 (sequences)

Stream 을 이해하는 데에 쓰이는 다음과 같은 유용한 비유가 있습니다 : Stream 2 개의 구성요소의 나열이라고 할 수 있습니다 : 이 2 개의 요소는 이전순서에 해당하는 요소와 다음순서에 해당하는 요소를 말합니다. 이런 모델에 대한 이해는 대단히 중요한데, 왜냐하면 스몰토크에서 모든 Stream 연산은 이 모델을 기본으로 하고있기 때문입니다. 이때문에 대부분의 Stream 클래스들은 PositionableStream의 서브클래스가 됩니다. 그림 10.1 은 5 개의 Character 를 포함하고 있는 Stream 을 보여주고 있습니다. 이경우, Stream은 처음의 위치에 있으며 이전요소는 가지고 있지 않습니다 (stream 내의 position 을 의미합니다). 사용자는 reset이라는 메시지를 사용해서 이 위치를 초기화 시킬 수 있습니다.

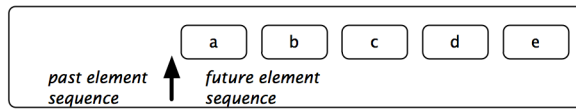


그림 10.1: 시작 시점에, 위치를 잡은 stream

요소를 읽는다 라는 행동을 개념적으로 본다면, 다음순서 요소를 떼어낸다음 그걸 이전요소의 뒷부분에 붙인다는것을 말합니다[1].

next 메시지를 사용해서 하나의 구성요소를 읽은 후, Stream의 상태는 그림 10.2 처럼 변경됩니다.

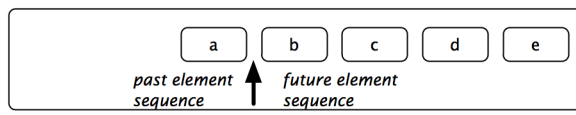


그림 10.2: 그림 10.1 의 stream에 next 메시지를 송신한 다음의 상태. chracter a는 이전에 속하지만 b, c, d, e는 다음에 속하게 된다.

요소를 작성*write* 한다는 것은, 현재 위치에 대한 다음 시퀀스의 앞쪽에 새로운 요소를 교대시키고, 이전 요소의 위치로 새로 바뀐 요소를 이동시킨다는 의미가 됩니다. 그림 10.3 은 메시지 nextPut:anElement 를 사용하여 x 를 작성*write* 한 이후의 stream의 상태를 보여주고 있습니다.

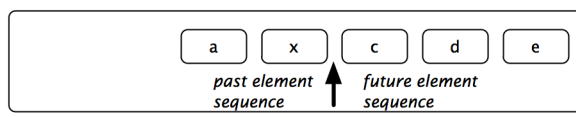


그림 10.3: x를 write 작업한 이후의 stream

[1] 지금 현재의 position을 기준으로 다음의 element 를 가져와서 이전 element의 뒤쪽으로 결합한다는 의미입니다. 결국 1:4 인 배열을 2:3 으로 바꾼다라는 의미가 되겠습니다.

10.2 Streams vs. Collections

컬렉션 프로토콜은 컬렉션의 요소에 대한 저장*storage*, 제거*removal* 그리고 열거형*enumeration* 등의 작업을 지원하지만, 이러한 연산들을 혼합하는걸 허용하지는 않습니다. 예를 들어, 만약 OrderedCollection의 구성요소가 do: 메소드를 이용해서 처리하는경우, do: 블록 내부에서 요소의 추가 또는 제거 작업은 불가능합니다. 그리고 컬렉션 프로토콜은, 컬렉션을 선택하며 선택된 컬렉션은 계속 진행하고 선택되지 않은 컬렉션은 진행을 멈추는 작업처럼, 동시에 2 개의 컬렉션을 반복 적용하기 위한 방법들을 제공하지 않습니다. 컬렉션 프로토콜은, 어떤 컬렉션이 계속 앞으로 진행되고 어떤 컬렉션이 그렇지 않은지를 선택함으로써 동시에 2 개의 컬렉션을 반복 적용하기 위한 방법들을 제공하지 않습니다. 이런 작업을 하고싶다면 컬렉션 내부의 index 또는 위치에 대한 참조를 콜렉션의 외부에서 관리할 필요가 있습니다: `ReadStream`, `WriteStream` 과 `ReadWriteStream` 은 정확히 이런경우에 사용할 수 있습니다.

바로 위에서 언급한 3 가지 클래스는 컬렉션을 stream 처리하기위해 만들어졌습니다. 예를들어 아래의 코드는 interval에 stream 을 만들고 2 개의 요소를 읽어들입니다.

```
r := ReadStream on: (1 to: 1000).
r next.      ⟶     1
r next.      ⟶     2
r atEnd.      ⟶     false
```

WriteStream은 컬렉션에 데이터를 쓰기*write* 할 수 있습니다.

```
w := WriteStream on: (String new: 5).
w nextPut: $a.
w nextPut: $b.
w contents.      ⟶     'ab'
```

읽기*read* 프로토콜과 쓰기*write* 프로토콜 양쪽을 지원하는 ReadWriteStreams 를 만드는 것도 가능합니다.

WriteStream와 ReadWriteStream의 주요 문제는, 스퀵의 경우 array와 string 을 지원한다는 것입니다. 이 문제는 Nile이라는 이름의 새로운 라이

브러리를 개발함으로서 개선을 진행중입니다만, 현시점에서는 다른 종류의 컬렉션에 stream 처리를 시도한다면, 에러가 납니다:

```
w := WriteStream on: (OrderedCollection new: 20).
w nextPut: 12.      ⟶      raises an error
```

stream은 컬렉션 뿐만 아니라 파일 그리고 소켓(network)에도 사용할 수 있습니다. 아래의 예제에서는 test.txt 라 작명된 캐리지 리턴으로 분리된 파일을 만들고 그 파일에 문자열을 작성한 다음 그 파일을 닫고 있습니다.

```
StandardFileStream
  fileNamed: 'test.txt'
  do: [:str | str
        nextPutAll: '123';
        cr;
        nextPutAll: 'abcd'].
```

다음장에서는 이런 프로토콜들을 좀 더 심도있게 알아보겠습니다.

10.3 컬렉션에 대한 stream 처리

stream은 컬렉션을 다루는데 매우 유용합니다. stream은 컬렉션에서 요소들을 읽고 쓰는 작업을 위해 사용할 수 있습니다. 이제부터 컬렉션을 다룰때 필요한 stream의 특징을 자세히 알아보도록 하겠습니다.

컬렉션 읽기

여기서는 컬렉션에 대한 읽기작업을 할 때 사용되는 기능을 소개하겠습니다. 컬렉션을 읽는 작업에 stream 을 사용한다는것은, 본질적으로는 컬렉션의 포인터(요소에 대한 Pointer)를 얻는다는것을 의미합니다. 이 포인터는 읽기를 하면 다음 방향으로 이동되며, 그 포인터를 사용자가 원할 때 원하는 위치로 이동시킬 수도 있습니다. ReadStream 클래스를 사용해서 컬렉션으로부터 요소들의 읽기작업을 할 수 있습니다.

next 메서드를 이용하면 컬렉션으로 부터 요소를 1 개 읽을 수 있으며, next: 메서드를 이용하면 원하는 개수만큼의 요소를 읽어낼 수 있습니다.

```
stream := ReadStream on: #(1 (a b c) false).
stream next.     ⟶     1
stream next.     ⟶     #(#a #b #c)
stream next.     ⟶     false
```

```
stream := ReadStream on: 'abcdef'.
stream next: 0.     ⟶     ''
stream next: 1.     ⟶     'a'
stream next: 3.     ⟶     'bcd'
stream next: 2.     ⟶     'ef'
```

peek 메시지는 앞으로 진행하지 않은 채, stream에서 다음 구성요소가 무엇인지 알고싶을때 사용합니다.

```
stream := ReadStream on: '--143'.
negative := (stream peek = $--). `look at the first element without
    reading it'
negative.     ⟶     true
negative ifTrue: [stream next].
`ignores the minus character'
number := stream upToEnd.
number.     ⟶     '143'
```

위의 코드는 stream에 숫자에 따라 얻어진 결과값이 $-$라면 boolean으로 값을 반환하고 결과값을 무조건 양수로 반환합니다. upToEnd 메서드는 현재 위치부터 stream의 끝까지 있는 모든것을 반환하며, 위치를 stream의 마지막 지점으로 설정합니다. 위의 코드는 peekFor: 를 사용해서 단순화 할 수 있으며, peekFor: 메서드는 stream의 다음차례의 요소와 인수가 같은 경우 stream의 위치를 그 다음으로 이동시키며 진행하고, 인수와 다를때는 stream의 위치를 이동시키지 않습니다.

```
stream := '--143' readStream.
(stream peekFor: $-)     ⟶     true
stream upToEnd     ⟶     '143'
```

peekFor: 는 또한 인수가 요소와 동일한 경우, 그 결과를 Boolean 으로 반환합니다.

위의 예제에서 stream 을 만드는 새로운 방법을 눈치채셨나요 : 순차 컬렉션에 대해 단지 readStream 을 보내는것 만으로도, 대상이 되는 컬렉션에 대한 read stream 을 만들 수 있습니다.

Positioning(위치선정). stream에 대한 포인터의 위치를 설정하기 위한 메서드가 있습니다. 만약 index 를 알고 있으면 position: 메서드를 사용해서 해당되는 위치로 직접 이동할 수 있죠. position 을 이용하면 현재의 위치를 알 수 있습니다. stream의 position은 요소 자체가 아니라 2 개의 요소 사이라는것을 꼭 기억해주시기 바랍니다. stream의 초기 index 상 위치는 0 입니다.

사용자는 아레의 코드를 이용해서 그림 10.4 에 그려진 stream의 상태를 얻을 수 있습니다.

```
stream := 'abcde' readStream.
stream position: 2.
stream peek     ⟶     $c
```

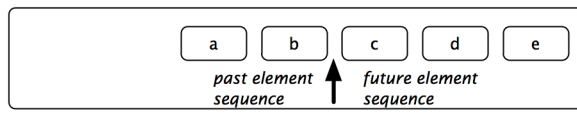


그림 10.4: 위치 2에 있는 스트림

stream 을 처음이나 맨 끝으로 이동시키려면, reset 또는 setToEnd 를 사용하면 됩니다. skip: 과 skipTo: 는 현재의 위치를 기준으로 다음위치로 이동하는 경우 사용합니다. skipTo: 는 건네받은 인수와 같은 요소를 찾을때까지 stream에 있는 모든 요소를 건너뛰는 반면에, skip: 인 인수로 받은 숫자만큼 요소를 건너뜁니다. 매칭된 요소의 뒤쪽에 stream의 포인터가 위치한다는걸 잊지 마시기 바랍니다.

```
stream := 'abcdef' readStream.
stream next.                                ⟶     $a    `stream
    is now positioned just after the a'
stream skip: 3.                                      `stream is
    now after the d'
stream position.                            ⟶     4
```

```
stream skip: ----2.                                          `stream is
    after the b'
stream position.                          ⟶     2
stream reset.
stream position.                          ⟶     0
stream skipTo: $e.                                   `stream is
    just after the e now'
stream next.                               ⟶     $f
stream contents.                          ⟶     'abcdef'
```

보시는 것처럼 글자 e를 건너뛰었습니다.

contents 메서드는 항상 전체 stream의 복사본을 반환합니다.

Testing. 몇가지 메서드를 이용하면 현재의 stream 상태를 테스트 할 수 있습니다: isEmpty는 컬렉션이 가지고있는 요소가 없는경우 true 를 반환하는 반면, atEnd는 컬렉션의 더이상 읽을 수 있는요소가 없는 경우 true 를 반환합니다.

알고리즘에서 atEnd 를 사용하는 경우를 살펴보도록 하겠습니다. 아래의 코드에서는 2 개의 정렬이 끝난 콜렉션을 인자로 받아, 이것들을 병합해서 정렬이 끝난 새로운 콜렉션으로 만듭니다.

```
stream1 := #(1 4 9 11 12 13) readStream.
stream2 := #(1 2 3 4 5 10 13 14 15) readStream.

`The variable result will contain the sorted collection.'
result := OrderedCollection new.
[stream1 atEnd not & stream2 atEnd not]
  whileTrue: [stream1 peek < stream2 peek
    `Remove the smallest element from either stream and add it to
    the result.'
   ifTrue: [result add: stream1 next]
   ifFalse: [result add: stream2 next]].

`One of the two streams might not be at its end. Copy whatever
    remains.'
result
  addAll: stream1 upToEnd;
  addAll: stream2 upToEnd.

result.     ⟶     an OrderedCollection(1 1 2 3 4 4 5 9 10 11 12
    13 13 14 15)
```

컬렉션 쓰기

ReadStream 을 사용해서, 컬렉션의 요소를 반복 진행을 통해 읽는 방법을 위에서 살펴봤습니다. 이제부터는 WriteStreams 를 사용해서 컬렉션을 만드는 방법을 배워보도록 하겠습니다.

WriteStreams은 컬렉션 안의 다양한 위치에서 많은 데이터를 덧붙이는 작업에 유용합니다. 아래의 예제처럼, 정적이거나 동적인 문자열을 생성하는 경우에 주로 사용됩니다:

```
stream := String new writeStream.
stream
  nextPutAll: 'This Smalltalk image contains: ';
  print: Smalltalk allClasses size;
  nextPutAll: ' classes.';
  cr;
  nextPutAll: 'This is really a lot.'.

stream contents.     ⟶     'This Smalltalk image contains: 2322
    classes. This is really a lot.'
```

이런 기법은, 다양한 클래스의 printOn: 메서드의 구현에서 사용되고 있습니다. 단순히 String 을 생성해야하는 경우, stream 을 사용하게되면 조금 더 단순하고 효과적인 방법을 사용할 수 있습니다.

```
string := String streamContents:
  [:stream |
    stream
    print: #(1 2 3);
    space;
    nextPutAll: 'size';
    space;
    nextPut: $=;
    space;
    print: 3. ].
string.     ⟶     '#(1 2 3) size = 3'
```

메서드 `streamContents:` 는 컬렉션과 컬렉션을 이용한 stream 을 만듭니다. 이 메서드는 인자로 블록을 받아 실행하며, 인자로 주어질 블록에는 stream 이 들어가야 합니다. 블록이 종료되면, streamContents:는 컬렉션의 내용을 반환합니다.

다음 WriteStream 메서드는 이런 컨텍스트에서 특별히 유용합니다:

nextPut: 인자로 받은 내용을 stream에 추가합니다.

nextPutAll: 인자로 받은 컬렉션의 각 구성요소를 stream에 추가합니다.

print: 인자의의 텍스트표현을 stream에 추가합니다.

특수문자를 stream에 표시하는데 편리한 `space`, `tab` 그리고 CR*carriage return* 등의 메서드도 있습니다. 또 다른 유용한 메서드는, 마지막 문자가 공백이 아닐 경우 공백을 추가해서, stream에서 마지막 문자가 공백이 되도록 보장하는 `ensureASpace` 입니다.

문자(열)의 결합에 대하여. WriteStream에서 문자(열)을 연결하는 작업에 있어 가장 좋은 방법은 nextPut:, nextPutAll: 을 사용하는 것입니다. 콤마연산자(,) 를 이용한 문자(열)의 결합은 메서드를 사용하는것에 비교하면 상당히 비효율적인 방식입니다:

```
[| temp |
  temp := String new.
  (1 to: 100000)
    do: [:i | temp := temp, i asString, ' ']] timeToRun     ⟶
      115176 `(milliseconds)'

[| temp |
  temp := WriteStream on: String new.
  (1 to: 100000)
    do: [:i | temp nextPutAll: i asString; space].
  temp contents] timeToRun    ⟶    1262 `(milliseconds)'
```

stream 을 사용하는 것이 보다 효율적인 이유는, 콤마연산자는 수신자와 인수를 연결하는 새로운 문자열을 만들기때문에, 수신자와 인수의 양쪽 모두를 복사하는 작업을 진행하게 되는 상황은 비효율적이기 때문입니다.

만약 같은 수신자에 대해서 콤마연산자로 연결을 반복하면 연결할때마다 문자열이 길어지기때문에, 복사해야하는 문자의 개수가 기하급수적으로 증가하게 되겠죠.

그리고 이런식으로 콤마연산자를 이용한 문자열의 연결을 반복하게되면 메모리상에 쓰레기를 더 많이 남기게 되며, garbege collection의 대상이 됩니다. 콤마연산자를 이용한 문자(열)의 연결 대신 stream 을 사용하는 방법은 유명한 최적화 방법입니다. 실제로 이런 작업을 위해서 streamContents[2] 사용하실 수 있습니다:

```
String streamContents: [ :tempStream |
  (1 to: 100000)
    do: [:i | tempStream nextPutAll: i asString; space]]
```

동시에 읽고 쓰기

콜렉션에 접근해서 동시에 읽고 쓰기를 하려는경우 stream 을 사용하면 이런 작업이 가능합니다. web browser에서 앞 버전과 뒤 버튼을 관리하기 위해 History 클래스를 만든다고 가정해 보겠습니다. History 클래스는 그림 10.5 에서 10.11까지의 내용과 같은 동작을 하게 될겁니다.

그림 10.5: 새로운 History가 비었습니다. 웹브라우저에 표시한 내용은 아무것도 없습니다.

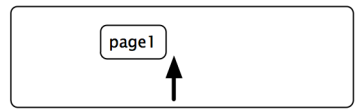


그림 10.6: 사용자는 1페이지를 엽니다.

위의 그림같은 동작은 ReadWriteStream 을 사용하여 구현할 수 있습니다.

```
Object subclass: #History
  instanceVariableNames: 'stream'
```

[2] 223페이지에서 확인가능

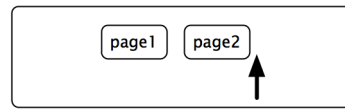


그림 10.7: 사용자는 2페이지에 대한 링크를 클릭합니다.

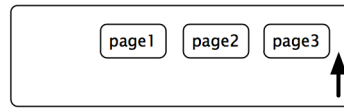


그림 10.8: 사용자는 3페이지에 대한 링크를 클릭합니다.

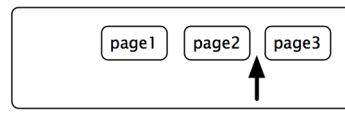


그림 10.9: 사용자는 뒤로 가기 버튼을 클릭합니다. 2페이지를 다시 보고 있습니다.

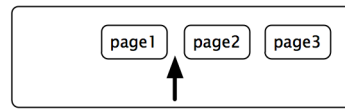


그림 10.10: 사용자는 뒤로 가기 버튼을 다시 클릭합니다. 1페이지가 지금 나타났습니다.

그림 10.11: 1페이지에서 부터, 사용자는 4페이지에 대한 링크를 클릭합니다. History는 2페이지와 3페이지를 잊어버립니다.

```
  classVariableNames: ''
  poolDictionaries: ''
  category: 'SBE----Streams'
```

```
History>>initialize
  super initialize.
  stream := ReadWriteStream on: Array new.
```

여기에서 딱히 어려운 부분은 없습니다. 이제 stream 을 포함하는 새로운 클래스를 정의하도록 하겠습니다. stream은 initialize 메서드를 통해 만들어집니다.

그 다음 앞 과 뒤 양쪽으로 이동하기 위한 메서드가 필요하겠죠:

```
History>>goBackward
  self canGoBackward ifFalse: [self error: 'Already on the first
    element'].
  stream skip: ----2.
  ↑ self next.

History>>goForward
  self canGoForward ifFalse: [self error: 'Already on the last
    element'].
  ↑ stream next
```

지금까지의 코드는 꽤 간단합니다. 이제 유저가 링크를 클릭했을 때, 동작해야할 goTo: 메서드를 작업하도록 하겠습니다. 아래와 같은 방법도 가능합니다:

```
History>>goTo: aPage
  stream nextPut: aPage.
```

하지만 지금까지의 작업이 완전한것은 아닙니다. 왜냐하면 유저가 링크를 클릭한 이후, 내부에서 앞 에 해당하는 페이지가 있어서는 안되기 때문입니다. 예를들어 앞 에 해당하는 버튼은 비활성화가 되어야 한다는거죠. 이런 작업에 대한 가장 단순한 해결책은 nil 을 사용해서 history의 끝을 나타내면 됩니다.

```
History>>goTo: anObject
  stream nextPut: anObject.
  stream nextPut: nil.
  stream back.
```

이제 canGoBackward와 canGoForward의 구현만 남았군요.

stream은 항상 2 개의 요소 사이에 위치하게 됩니다. 뒤로 가는 작업을 하기위해서는 현재의 위치 앞쪽으로 2 개의 페이지가 있어야 합니다. 하나는 현재의 페이지고, 다른 하나는 이동을 원하는 페이지가 되겠죠.

```
History>>canGoBackward
  ↑ stream position > 1

History>>canGoForward
  ↑ stream atEnd not and: [stream peek notNil]
```

stream의 내용을 확인하는 메서드를 추가해 보도록 하겠습니다:

```
History>>contents
  ↑ stream contents
```

History는 예상대로 작동합니다:

```
History new
  goTo: #page1;
  goTo: #page2;
  goTo: #page3;
  goBackward;
  goBackward;
  goTo: #page4;
  contents      ⟶      #(#page1 #page4 nil nil)
```

10.4 파일 접근에 대한 stream의 활용

컬렉션을 stream 으로 처리하는 방법은 이미 배웠습니다. 동일한 방법으로 하드디스크에 있는 파일도 stream 으로 처리할 수 있습니다. 일단 만들어지만, 파일에 대한 stream은 콜렉션에 대한 stream 과 거의 비슷합니다: 같은 프로토콜을 이용해서 stream에 대한 읽기, 쓰기, 위치결정 등의 작업을 할 수 있습니다. 가장 큰 차이는 stream 을 만드는 방법에 있습니다. 파일에 대한 stream 을 만드는 방법에 대해서, 몇가지 서로 다른 방법을 살펴보겠습니다.

파일 stream 만들기

파일 stream 을 만들기 위해서는, FileStream 클래스 에서 제공되는 아래의 인스턴스 생성 메서드들 중 하나를 반드시 사용해야 합니다:

fileNamed: 주어진 이름으로 파일을 읽기, 쓰기 작업을 위해서 open 힙니다. 만약 파일이 이미 존재하는 경우, 이전의 내용은 변경하거나 옮기는게 가능합니다만, close 하는 경우 파일이 불완전한 상태가 되지는 않습니다. 만약 인수로 받은 이름에 별도로 디렉토리에 대한 내용이 포함되어 있지 않다면 파일은 기본디렉토리에서 생성됩니다.

newFileNamed: 주어진 이름으로 새로운 파일을 만들고, 그 파일에 대해 쓰기 작업을 위한 stream 을 반환합니다. 만약 파일이 이미 존재한다면, 유저에게 해야할 일을 요청합니다.

forceNewFileNamed: 주어진 이름으로 새로운 파일을 만들고, 그 파일에 대해 쓰기 작업을 위한 stream 을 반환합니다. 만약 파일이 이미 존재한다면, 질문없이 해당파일을 삭제하고 새로운 파일을 생성합니다.

oldFileNamed: 읽기와 쓰기 작업을 위해 주어진 이름으로 기존에 존재하는 파일을 open 합니다[3]. 만약 파일이 이미 존재하는 경우, 이전의 내용은 변경하거나 옮기는게 가능합니다만, close 하는 경우 파일이 불완전한 상태가 되지는 않습니다. 만약 인수로 받은 이름에 별도로 디렉토리에 대한 내용이 포함되어있지 않다면 파일은 기본디렉토리에서 생성됩니다.

readOnlyFileNamed: 읽기 작업만을 위해 주어진 이름으로 이미 존재하고 있는 파일을 엽니다.

open된 파일 stream은 꼭 닫아주어야 합니다. 이런 닫기작업에는 `close` 메서드를 사용합니다.

[3] Pharo by example 일본어판 에서는 파일이 실제로 존재하지 않는경우에 대한 동작을 사용자에게 질문한다고 되어있습니다. 아마도 squeak by example에서는 당연한 action이라고 생각하고 언급을 안한거같기는 하군요

```
stream := FileStream forceNewFileNamed: 'test.txt'.
stream
  nextPutAll: 'This text is written in a file named ';
  print: stream localName.
stream close.

stream := FileStream readOnlyFileNamed: 'test.txt'.
stream contents.      ⟶      'This text is written in a file
    named \emph{test.txt}'
stream close.
```

메서드 `localName` 은 파일 이름의 마지막 요소를 반환합니다. 메서드 `fullName` 을 이용해서 전체 경로를 얻을 수도 있습니다.

아마도 당신은 머지않아 파일 stream 을 수동으로 닫는것이 꽤나 번거롭고, 에러가 발생될 수도 있다는걸 알게 될겁니다. 이런 문제때문에 `FileStream` 클래스는 `forceNewFileNamed:do:` 라는 메시지를 제공해서 작성 작업을 블록형태의 인자로 받아서 작성 작업을 끝낸후, 자동으로 stream 을 close 합니다.

```
FileStream
  forceNewFileNamed: 'test.txt'
  do: [:stream |
    stream
      nextPutAll: 'This text is written in a file named ';
      print: stream localName].
string := FileStream
      readOnlyFileNamed: 'test.txt'
      do: [:stream | stream contents].
string      ⟶      'This text is written in a file named
    \emph{test.txt}'
```

파일 stream 을 만드는 메서드중 블록을 인자로 취하는 경우는, 일단 stream 만든 후, stream 을 인자로 블록을 실행하고 마지막에 stream 을 close 합니다. 이런 종류의 메서드는 블록에서 반환되는 내용을 반환하게 되며, 이것은 곧 블록의 마지막에 있는 표현식의 값이 됩니다. 위의 예제에서는 블록의 마지막 표현식은 stream contents 를 실행해서 파일의 내용을 얻은다음, 그 내용을 string이라는 변수에 대입하고 있습니다.

Binary Stream

기본적으로 파일 stream은, text 기반으로서 chracter의 읽기, 쓰기 작업용으로 만들어집니다. 만약 stream에서 binary 데이터를 취급해야한다면, 반드시 stream에 binary 메시지를 보내야 합니다.

stream이 binary mode 인 경우, 0 에서 255(1 byte)의 값만 사용할 수 있습니다. nextPutAll: 메서드를 이용해서 한번에 2 개 이상의 값을 쓰기*write* 하고싶다면 이 메서드의 인자로 ByteArray 를 넘겨주어야 합니다.

```
FileStream
  forceNewFileNamed: 'test.bin'
  do: [:stream |
    stream
      binary;
      nextPutAll: #(145 250 139 98) asByteArray].

FileStream
  readOnlyFileNamed: 'test.bin'
  do: [:stream |
    stream binary.
    stream size.        ⟶     4
    stream next.        ⟶     145
    stream upToEnd.     ⟶     a ByteArray(250 139 98)
  ].
```

아래의 예제에서 "test.pgm" 이라고 하는 그림파일을 만들어 보도록 하겠습니다. 작성한 파일은 원하는 draw 프로그램으로 열 수 있습니다[4].

```
FileStream
  forceNewFileNamed: 'test.pgm'
  do: [:stream |
    stream
      nextPutAll: 'P5'; cr;
      nextPutAll: '4 4'; cr;
      nextPutAll: '255'; cr;
      binary;
      nextPutAll: #(255 0 255 0) asByteArray;
      nextPutAll: #(0 255 0 255) asByteArray;
      nextPutAll: #(255 0 255 0) asByteArray;
      nextPutAll: #(0 255 0 255) asByteArray
```

[4] pharo by example 일본어판 에서는 PGM:Portablegraymap이라는 내용이 있습니다. GIMP 등의 프로그램에서 이 포맷의 사용이 가능합니다.

```
]
```

위의 프로그램으로 그림 10.12 와 같은 4×4 의 체스판을 만들 수 있습니다.

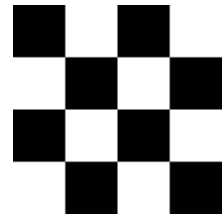


그림 10.12: binary stream 을 사용해서 4x4 체스판을 그릴 수 있습니다.

10.5 10장 요약

stream은 요소의 시퀀스(일련의 구성요소)를 순차적으로 읽고, 쓰는 작업에 대해서 컬렉션보다 더 나은 방법을 제공합니다. stream 과 컬렉션은 서로 변환을 하려는경우 간단한 방법을 제공합니다.

- stream은 읽기 또는 쓰기 작업이 가능하거나 한번에 읽기 쓰기 작업이 모두 가능합니다.
- 컬렉션을 stream 으로 변환하려 하는경우, 컬렉션에 "on" stream 을 정의합니다. 예를 들어, `ReadStream on: (1 to:1000)` 또는 메시지 readStream 등을 컬렉션에 보내면 됩니다.
- stream 을 컬렉션으로 변환하려 하는경우, 메시지 contents 를 보내면 됩니다.
- 요소의 개수가 많아서 크기가 큰 컬렉션들을 연결하려는 경우, 콤마 연산자 보다는, stream 을 만들고 컬렉션을 nextPutAll: 메서드로 stream 에 추가하고 contents 를 전송해서 결과를 얻어내는 것이 보다 더 효율적입니다.

- 파일 stream은 기본적으로, text 기반입니다. 파일 stream 을 binary 로 만들기 위해서 파일 stream에 binary를 반드시 전송해야 합니다.

제 11 장

Morphic

Morphic 은 스퀵의 그래픽 인터페이스에 붙여진 이름입니다. Morphic은 스몰토크로 작성되었기 때문에, 여러가지 OS 사이에서 완전하게 사용가능하며 그 결과 스퀵의 모양새는 Unix, MacOS 그리고 Windows에서 완전히 동일합니다. 대부분의 다른 유저 인터페이스 도구 kit 과 Morphic 과의 가장 큰 차이점은, Morphic은 인터페이스를 "구성" 하고 "실행" 하기 위한 모드를 나눠서 취급하지 않는다는 점입니다: 화면상의 모든 그래픽 구성요소들은 언제라도 사용자가 조립하거나 분해할 수 있습니다.[1]

11.1 Morphic의 역사

Morphic은 John Maloney와 Randy Smith가 Self 라는 프로그래밍 언어[2]로 1993년경 개발을 시작하였습니다. Maloney는 나중에 스퀵을 위한 새로운 버전의 Morphic 을 만들었습니다만 self 프로그래밍 언어버전에서 가지고있던 *직접성*directness 과 *생동감*liveness 등의 기본 아이디어는 여전히 살아 있

[1] 우리는 Hilaire Fernandes가 이 장의 기초를 이루는 프랑스어로 된 그의 원 저술의 사용을 허락해 준 것에 대해 감사드립니다.

[2] Self programming Language에 대한 내용은 http://en.wikipedia.org/wiki/Self_programming_language에서 찾을 수 있습니다.

으며, 스퀵Morphic에 잘 반영되어 있습니다. 직접성이란 화면에 있는 모양을 직접 검사하고 변경할 수 있다는 것을 의미하며, 이런 작업들은 마우스를 이용한 포인팅으로 가능합니다. 생동감이란 유저 인터페이스는 언제라도 유저의 조작에 반응할 수 있다는 것을 의미합니다: 즉 화면상의 정보는 정보를 보여주고 있는 world가 변경될때마다 자동으로 갱신 됩니다. 간단한 예로서, 메뉴에서 항목을 떼어내서 버튼으로 사용할 수 있습니다.

World 메뉴를 불러옵니다. *Morphic* 할로를 불러내기 위해 *World* 메뉴에서 한번 파랑 클릭을 하고, 할로를 불러내기 위해, 따로 분리하기 원하는 메뉴 아이템을 다시 한번 파랑 클릭하십시오. 이제 그림 *11.1* 에 보이는 것 처럼 검정색 손잡이를 붙잡아 화면 위의 원하는 장소에 끌어다 놓아주세요.

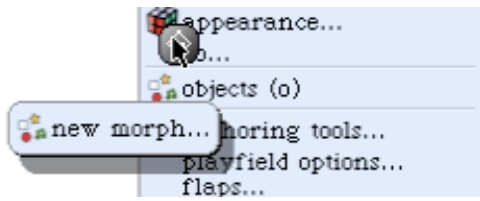


그림 11.1: `new morph`라는 메뉴 아이템을, 독립된 버튼으로 만들기 위해 `morph`를 떼어 분리합니다.

스퀵을 실행한 후 화면에 보이는 모든 객체는 Morph이며, 이런 객체들은 Morph 클래스의 인스턴스입니다. Morph 클래스는 많은 메서드를 가지고 있는 큰 클래스이며, 작은 코드로도 재미있는 동작을 하는 서브클래스를 만들 수 있십니다. 모든 객체는 moprh 로 만들 수 있습니다만, 어느정도 좋은모양을 가지는지는 객체 자체에 달렸습니다.

String 객체를 표시하기 위해 *Morph* 를 만드려면, 워크스페이스에서 한번에 한 줄씩 다음 코드를 실행하십시오.

```
s := 'Morph' asMorph openInWorld.
s openViewerForArgument
```

첫 번째 줄은, 문자열 `'Morph'`를 world에 나타내가 위해 Morph를 만들고, 스퀵에서 부르는 이름인 "world" 라는 객체에서 만들어진 Morph를 엽니

다(만들어진 morph를 표시한다는 거죠). 사용자는 파랑 클릭으로 조작이 가능한 그래픽 구성요소-Morph-를 얻을 수 있어야만 합니다. 두 번째 라인은 화면에서 Morph의 x와 y 좌표와 같은 이 Morph의 내부를 보여주는 "viewer" 를 엽니다. 노랑 느낌표중 하나를 클릭하면 메시지를 Morph에 전송하며, 전송한 메시지에 알맞는 결과를 얻을 수 있십니다.

물론, 지금 확인한 그래픽 표현보다 좀 더 재미있는 그래픽 표현을 가지는 Morph를 정의하는것도 가능합니다. class Object 클래스에 있는 asMorph 메서드의 기본값은 단지 StringMorph 를 만들기만 합니다. 예를 들어, `Color tan asMorph`, 단지 Color `tan printString`의 실행결과를 라벨로 가지는 StringMorph를 반환합니다. 그러면 이것을 변경해서 색상이 지정되는 직사각형이 반환되도록 해보겠습니다.

 *Color class*에서 브라우저를 열고 다음 메서드를 추가하십시오:

Method 11.1: `Color`의 인스턴스를 위해 `Morph` 얻기

```
Color>>asMorph
    ↑ Morph new color: self
```

워크스페이스에서 `Color orange asMorph openInWorld` 를 실행합니다. String의 Morph 대신에 오렌지색 직사각형을 얻을 수 있습니다!

11.2 morph의 조작

morph는 객체이기 때문에, 스몰토크에서 다른 모든 객체를 다루듯이 조작할 수 있습니다: 메시지를 보내는 것으로 morph의 속성을 바꿀 수 있고 morph의 서브클래스를 만드는 작업등을 할 수 있습니다. 모든 morph는 현재 화면에 열려있지 않은 상태여도, 위치와 크기를 가지고 있습니다. 모든 morph는 편의상, 화면에서 사각형의 영역을 가지고있는것으로 가정되고, 규칙적인 모양이 아니라고 해도, 위치와 크기의 값은 대상이되는 morph 를 둘러싸는 최소크기의 직사격형 상자가 되며, 이 상자를 morph의 bounding box 또는 "bounds" 라고 합니다. Postion 메서드는 morph의 상단 좌측 모서리의

위치를 기술하는 점(point)의 값을 반환합니다. morph에 대한 좌표체계의 기준점은 화면 좌상단 구석에 있으며, 화면에서 하단으로 증가하는 경우는 y 좌표, 화면에서 우측으로 증가하는 경우는 x 좌표가 됩니다. extent 메서드도 postion 메서드처럼 point 를 반환합니다만, 위치값을 반환하는것이 아니라 morph의 폭과 높이에 대한 값을 반환합니다.

 다음 코드를 워크스페이스에 입력한 후 do it 을 실행하십시오.

```
joe := Morph new color: Color blue.
joe openInWorld.
bill := Morph new color: Color red .
bill openInWorld.
```

joe position 을 입력한 후 print it 을 실행합니다. joe 를 움직이기 위해서, joe position: (joe position + (10@3)) 을 반복적으로 실행합니다.

크기에 대해서도 같은 작업을 할 수 있습니다. joe extent는 joe의 크키를 반환하며, joe의 크기를 크게 하려면, joe extent: (joe extent * 1.1) 를 실행합니다. Morph의 색상을 변경하려면, 대상이 되는 morph에 인자로 전달을 원하는 Color 객체와 함께 color: 메시지를 전송합니다. 예를 들어, joe color: Color orange 같은 내용이 되겠군요. 투명도를 추가하려면 joe color: (Color orange alpha: 0.5) 를 시도해보세요.

 *bill*이 *joe*를 따르게 만들려면, 이 코드를 반복적으로 실행하십시오.

```
bill position: (joe position + (100@0))
```

만약 마우스를 써서 joe 를 이동시키고 이 코드를 실행한다면, bill은 joe의 우측 100 pixel 로 이동할겁니다.

11.3 morph의 합성 (Composing Morphs)

새로운 그래픽 표현을 만드는 한 가지 방법은 하나의 morph 를 다른 morph 안으로 이동하는 것입니다. 이것을 합성*composition* 이라고 하며 morph는 어떤

depth 라도 이동이 가능합니다. 사용자는 메시지 addMorph: 를 container morph에 전송하여 morph 를 다른 morph 안쪽으로 이동시킬 수 있습니다.

Morph 를 다른 *morph*에 추가하는 작업을 시도해 보십시오.

```
star := StarMorph new color: Color yellow.
joe addMorph: star.
star position: joe position.
```

위 예제에서 마지막줄은 별을 joe와 동일한 좌표로 이동시킵니다. 안쪽으로 포함된 morph의 좌표는, 그릇이 되는 morph가 기준이 되는것이 아니라 여전히 화면과 관련이 있다는걸 주의해 주시기 바랍니다. morph의 위치를 설정하기 위한 많은 메서드가 있으며, Morph 클래스의 geometry 를 탐색*browse* 하면 원하는 morph의 좌표를 얻을 수 있습니다. 예를들어, joe 내부의 별을 joe의 중앙에 위치시키기 위해서 star center: joe center 를 실행해 보도록 합니다.

그림 11.2: 투명한 파랑 모프인 joe에 별을 포함하고 있습니다.

만약 마우스를 이용해서 별을 잡으려고 한다면, 실제로 joe 를 잡은것을 확인할 수 있으며, 두개의 morph는 함께 움직이게 됩니다: 별은 joe 내부에 들어가있게 됩니다. joe 내부에 더 많은 morph 를 포함시키는것도 가능하죠. 게다가 이런 작업을 프로그램적으로 진행하기 위해서, 직접적인 조작 만약 여러분이 마우스로 별을 쥐었다면, 실제로 joe를 쥐게 된 것임을 발견하게 되실것이며, 두 개의 모프는 함께 이동합니다: 별은 joe 내부에 삽입됩니다. joe 내부에 더 많은 모프를 삽입하는 것이 가능합니다. 게다가, 이런 경우는 프로그램으로 조작하는것 외에도, 직접 조작으로 morph의 삽입이 가능합니다.

world 메뉴에서 *"object"* 를 선택하고 *"Graphics"* 라벨이 붙은 버튼을 클릭하세요. Supplies 플랩으로부터 타원과 별을 드래그해주세요. 타원 위에 별을 이동시킨 후, 메뉴를[3] 불러내기 위해 별을 노랑 클릭합니다. embed into ▷ Ellipse 를 선택하십시오. 이제 별과 타원이 함께 움직이게 됩니다.

서브모프*sub-morph* 를 이동시키기 위해, joe removeMorph: star 또는 star delete 를 실행합니다. 이 작업은 직접조작으로도 가능합니다.

별을 두 번 파랑 클릭하십시오. *Grab* 손잡이를 사용하여 타원으로부터 별을 드래그 해내주세요.

첫 번째 클릭은 타원의 Morphic 할로를 불러옵니다. 두 번째 클릭은 별의 Morphic 할로를 불러옵니다. 한번 클릭할때마다 Morphic 할로를 불러올 대상이 한단계씩 내려갑니다.

11.4 나만의 morph 를 생성하고 그리기

Morph의 합성을 이용해서 많은 종류의 편리하고, 재미있는 그래픽 표현을 만들 수 있지만, 가끔은 완전히 다른 뭔가를 만들고 싶을때가 있겠죠. 그런 작업을 하기 위해서는, Morph의 서브클래스를 만들어서 drawOn: 메서드를 오버라이드*override* 해서 morph의 모양새를 바꿀 수 있습니다.

Morphic 프레임워크는 화면에 morph 를 다시 표시해야하는 경우, 메시지 `drawOn:` 을 morph에 보냅니다. drawOn: 메시지의 인자는, Canvas 클래스 (또는 서브클래스) 의 인스턴스로서, morph가 스스로를 canvas의 bounds[4] 내에 표시할거라는걸 예상할 수 있겠죠.

클래스 브라우저를 사용하여 `Morph` 로부터 상속받은 새로운 클래스 `CrossMorph`를 정의하십시오:

[3] 여러분은 또한 모픽 할로를 불러내기 위해 별을 파랑 클릭하고 빨강 메뉴 손잡이를 클릭할 수 있습니다.

[4] 이 경우는 그릴 morph의 bounds가 아니라 그림판이 될 대상 canvas의 bounds 를 의미하는것 같습니다

Class 11.2: *CrossMorph*를 정의하기

```
Morph subclass: #CrossMorph
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'SBE----Morphic'
```

아래와 같이 drawOn: 메서드를 정의할 수 있습니다:

Method 11.3: *CrossMorph* 그리기.

```
drawOn: aCanvas
  | crossHeight crossWidth horizontalBar verticalBar |
  crossHeight := self height / 3.0 .
  crossWidth := self width / 3.0 .
  horizontalBar := self bounds insetBy: 0 @ crossHeight.
  verticalBar := self bounds insetBy: crossWidth @ 0.
  aCanvas fillRectangle: horizontalBar color: self color.
  aCanvas fillRectangle: verticalBar color: self color
```

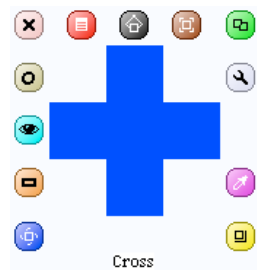


그림 11.3: `CrossMorph`가 할로를 표시한 모습. 원하는 만큼 `CrossMorph` 크기를 변경할 수 있습니다.

Morph에 `bounds` 메시지를 전송하면, Rectangle 클래스의 인스턴스인 bounding box가 반환되며, 이것은 morph가 Rectangle의 인스턴스 라는 의미가 됩니다. Rectangle은 geometry와 관련된 다른 사각형을 행성하는등의 많은 메시지를 사용할 수 있습니다; 위의 예제에서는, 인수로 점*point* 을 사용하는 `insetBy:` 메시지를 사용해서, 줄어든 높이로 첫번째 직사각형을 만들고, 줄어든 너비로 다른 직사각형을 만들고 있습니다.

새로운 *morph*를 테스트 하기 위해 `CrossMorph new openInWorld` 를 실행하십시오.

실행하면 그림 11.3 과 같은 결과를 얻을 수 있습니다. 그렇지만 마우스에 반응하는장소-morph 를 잡기위해 클릭할 수 있는 영역-는 bounding box 전체가 된다는걸 알아차렸을거라고 생각합니다. 이걸 고쳐보도록 하죠.

Morphic 프레임워크는, 어떤 morph가 마우스의 커서 아래에 있는지 알아내야할 필요가 있을때, 마우스의 포인터 아래에 bounding box가 있는 모든 morph에 대해서 containsPoint: 메시지를 보냅니다. 즉 마우스에 반응하는 영역을 morph의 십자 부분으로 제한하고 싶다면, containsPoint: 메서드를 오버라이드 할 필요가 있는거죠.

클래스 *CrossMorph*에서 다음 메서드를 정의하십시오 :

Method 11.4: `CrossMorph`의 반응 영역*sensitive zone* 설정하기

```
containsPoint: aPoint
    | crossHeight crossWidth horizontalBar verticalBar |
    crossHeight := self height / 3.0.
    crossWidth := self width / 3.0.
    horizontalBar := self bounds insetBy: 0 @ crossHeight.
    verticalBar := self bounds insetBy: crossWidth @ 0.
    ↑ (horizontalBar containsPoint: aPoint)
        or: [verticalBar containsPoint: aPoint]
```

이 메서드는 `drawOn:`과 동일한 로직을 사용하기 때문에, `containsPoint:`에서 `true`를 반환하는 영역이, drawOn에 의해 색칠이 가능한 영역과 일치한다는 것을 확신할 수 있죠. `Rectangle` 클래스의 `containsPoint:` 메서드를 활용하면, 이런 번거로운 작업을 해낼 수 있습니다.

메서드 **??** 과 11.4에 있는 코드는 2 가지 문제를 가지고 있습니다. 눈에 띄는 문제점으로는 코드가 중복되고 있다는거죠. 이건 매우 큰 잘못입니다 : 왜냐하면 `horizonatalBar` 또는 `verticalBar` 가 계산된 방식을 변경할 필요가 있을때, 2개 지점중 한쪽의 코드를 변경하는 작업을 쉽게 잊어버리기 때문입니다. 해결하려면 새로운 2개 메서드에서 계산에 관련된 부분을 분리해낸 후, `private` 프로토콜에 넣으면 됩니다.

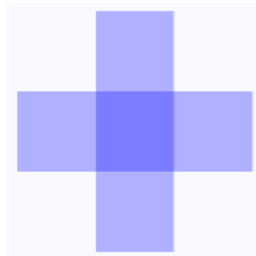


그림 11.4: colour로 2번 채워진 십자의 중심부분

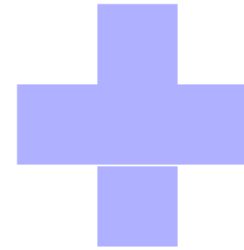


그림 11.5: 채워지지 않은 픽셀의 열을 보여주는 십자 모양의 Morph

Method 11.5: `horizontalBar`

```
horizontalBar
    | crossHeight |
    crossHeight := self height / 3.0.
    ↑ self bounds insetBy: 0 @ crossHeight
```

Method 11.6: `verticalBar`

```
verticalBar
    | crossWidth |
    crossWidth := self width / 3.0.
    ↑ self bounds insetBy: crossWidth @ 0
```

이렇게 만든 메서드를 이용해서`drawOn:`과 `containsPoint:` 양쪽을 다 정의하면 됩니다.

Method 11.7: 리팩토링한 `CrossMorph>>drawOn:`

```
drawOn: aCanvas
    aCanvas fillRectangle: self horizontalBar color: self color.
    aCanvas fillRectangle: self verticalBar color: self color
```

Method 11.8: 리팩토링한 `CrossMorph>>containsPoint:`.

```
containsPoint: aPoint
    ↑ (self horizontalBar containsPoint: aPoint)
        or: [self verticalBar containsPoint: aPoint]
```

private 메서드에 의미있는 이름을 붙인 덕분에, 이 코드는 보다 이해가 쉬워졌습니다. 이제 간단히 두번째의 문제를 알 수 있게 됐죠: 십자의 중앙부분이

horizontalBar와 verticalBar의 양쪽에 걸쳐져 있기때문에 2 번 그려지는 문제입니다. 이런 버그는 투명하지 않는 생상으로 십자를 그린다면 별 문제가 아닐겁니다만, 그림 11.4 에서 확인할 수 있듯이 반투명 십자를 그리는 순간, 확실히 버그가 드러날겁니다.

 다음 코드를 워크스페이스에서 한 라인씩 실행하십시오.

```
m := CrossMorph new bounds: (0@0 corner: 300@300).
m openInWorld.
m color: (Color blue alpha: 0.3).
```

이 문제를 해결하려면 verticalBar 를 세부분으로 나눠서, 위쪽과 아래쪽만 칠하면 됩니다. Rectangle 클래스에서 이런 여려운 일을 해주는 메서드를 다시 발견할 수 있습니다: r1 areasOutside: r2 는 r2 의 바깥쪽으로 삐져나온 부분에 해당되는 r1 을 구성하는 사각형의 배열을 반환합니다. 아래쪽에 수정된 코드가 있습니다:

Method 11.9: 개량된 *drawOn:* 메서드, 이 메서드는 십자의 중심부분을 한번만 채웁니다.

```
drawOn: aCanvas
  | topAndBottom |
  aCanvas fillRectangle: self horizontalBar color: self color.
  topAndBottom := self verticalBar areasOutside: self
    horizontalBar.
  topAndBottom do: [ :each | aCanvas fillRectangle: each color:
    self color]
```

위의 코드는 잘 작동하는걸로 보입니다만, 몇번정도 십자의 크기를 변경하려고 하는경우, 그림 11.5 처럼 특정 크기에서 1 pixel 만큼의 선이 십자의 아래부분과 위쪽을 나눠버리는것을 확인할 수 있습니다. 이 문제는 어림수(짐작하는수) 처리*rounding* 때문에 일어납니다: 색을 채우려고 하는 사각형의 크기가 정수*integer* 가 아닌경우, fillRectangle: color: 는 적당히 수치를 조절하기때문에 1 pixel 만큼 문제가 생기게 됩니다. 이 문제는, 사각형의 크기를 계산할때 명시적으로 값을 대입하는것으로 해결할 수 있습니다.

Method 11.10: 정확한 *CrossMorph»>horizontalBar rounding* 처리

```
horizontalBar
  | crossHeight |
  crossHeight := (self height / 3.0) rounded.
  ↑ self bounds insetBy: 0 @ crossHeight
```

Method 11.11: 정확한 *CrossMorph»>verticalBar rounding* 처리

```
verticalBar
  | crossWidth |
  crossWidth := (self width / 3.0) rounded.
  ↑ self bounds insetBy: crossWidth @ 0
```

11.5 상호 작용과 애니메이션

morph 모프를 사용해서 생동감있는 유저 인터페이스를 만들려면, 마우스와 키보드를 사용하여 인터페이스와 상호작용이 가능해야 합니다. 더군다나 morph는 스스로의 모양새와 위치-자신에 대한 애니메이션을 사용해서-를 변경함으로서 유저로 부터의 입력에 반응할 수 있어야 합니다.

마우스 이벤트

마우스 버튼을 눌렀을때, Morphic은 각 morph 를 대상으로 handlesMouseDown: 메시지를 전송합니다. 만약 morph가 true 를 반환하면, Morphic은 즉시 morph에게 mouseDown: 메시지를 보내며, 유저가 마우스 버튼에서 손을 뗐을때는 mouseUp: 메시지를 전송합니다. 만약 모든 morph가 false 를 반환하는경우, Morphic은 드래그앤드롭의 동작을 시작합니다. 아래에서 알아볼 내용에도 있겠지만, mouseDown: 과 mouseUp: 메시지는, MouseEvent 객체의 인자를 포함해서 전송됩니다.

CrossMorph 를 확장해서 마우스 이벤트를 취급할 수 있도록 해보겠습니다. 일단 crossMorphs의 모든 morph가 handlesMouseDown: 에 true 를 반환하게 하는것부터 시작하도록 하겠습니다.

 이 메서드를 `CrossMorph` 클래스에 추가하십시오.

Method 11.12: *CrossMorph*가 마우스 클릭에 반응하도록 선언합니다.

```
CrossMorph>>handlesMouseDown: anEvent
  ↑true
```

붉은 버튼을 클릭했을때 십자의 색상을 빨강으로 변경하고, 노랑 버튼이 클릭될때는 색상을 노랑으로 변경하기를 원한다고 가정해 보도록 하겠습니다. 이 작업은 메서드 11.13 을 이용해서 실행할 수 있습니다.

Method 11.13: 마우스 클릭에 반응해서 *morph*의 색상을 변경하기

```
CrossMorph>>mouseDown: anEvent
  anEvent redButtonPressed
    ifTrue: [self color: Color red].
  anEvent yellowButtonPressed
    ifTrue: [self color: Color yellow].
  self changed
```

이 메서드에서 morph의 색상을 변경한 다음 self changed 를 보낸다는것에 주의합시다. 이 self 부분 때문에 Morphic은 즉시 drawOn: 을 보내게 됩니다. 추가로, morph가 마우스 이벤트를 처리하면 유저는 더이상 morph를 마우스로 잡아서 이동시킬 수 없게된다는걸 유념해 주시기 바랍니다. 이동 등의 작업을 하고싶은 경우는 할로를 사용해야만 합니다 : 할로를 보이게 하기 위해서 morph 를 파랑 클릭하고 morph 상단에서 morph를 파랑 클릭하고, 모프 상단에서 갈색 이동 손잡이  과 검정색 픽업 손잡이  중 하나를 잡습니다.

mouseDown: 메서드의 인수인 anEvent는 MorphicEvent의 서브클래스인 MouseEvent의 인스턴스 입니다. MouseEvent에는 redButtonPressed와 yellowButtonPressed 메서드가 정의되어 있습니다. 마우스 이벤트를 취급하는 메서드는 어떤것들이 있는지 알고싶다면, MouseEvent 클래스를 살펴보면 됩니다.

키보드 이벤트

키보드 이벤트를 얻기위해, 다음의 3 가지 단계들을 거쳐야 합니다.

1. 특정 morph에 "keyboard focus" 를 부여합니다: 예를 들어,, 마우스가 morph 위에 있는경우 해당 morph에 focus 를 부여할 수 있습니다.

2. handleKeystroke: 메서드를 사용해서 morph가 키보드 이벤트를 처리하게 합니다:이 메시지는 유저가 키를 눌렀을때, 키보드 포커스를 가지고 있는 morph에 전송됩니다.

3. 마우스의 포인터가 더 이상 대상이 되는 morph 위에 위치하지 않는경우, keyboard focus 를 release 합니다.

CrossMorph 를 확장해서, 키입력에 반응하도록 만들겠습니다. 첫번째로, 마우스의 포인터가 대상이 되는 morph 위에 있는경우 알림을 받도록 해야할거같군요. 이 알림은, 만약 대상이 되는 morph가 handlesMouseOver: 메시지에 true 를 반환할때 발생합니다.

 마우스 포인터 아래에 있을 때, *CrossMorph*가 반응하도록 선언하십시오.

Method 11.14: *"mouse over"* 이벤트를 처리할 수 있게 합니다.

```
CrossMorph>>handlesMouseOver: anEvent
  ↑true
```

마우스 위치에 대한 처리는 handlesMouseDown: 과 같습니다. 마우스 포인터가 morph에 들어가거나 바깥으로 이동할때 mouseEnter:와 mouseLeave: 메시지가 morph 로 전달됩니다.

 *2* 개의 메서드를 정의해서 `CrossMorph` 가 *keyboard focus* 를 붙잡거나 놓고, 세 번째 메서드가 실제로 키보드입력을 취급하도록 정의하십시오.

Method 11.15: 마우스가 *morph*에 들어갈 때, *keyboard focus* 얻기

```
CrossMorph>>mouseEnter: anEvent
  anEvent hand newKeyboardFocus: self
```

Method 11.16: 마우스 포인터를 바깥쪽으로 이동시키면, 초점은 원래 있던 자리로 돌아갑니다.

```
CrossMorph>>mouseLeave: anEvent
  anEvent hand newKeyboardFocus: nil
```

Method 11.17: 키보드 이벤트를 수신하고 전달하기

```
CrossMorph>>handleKeystroke: anEvent
  | keyValue |
  keyValue := anEvent keyValue.
  keyValue = 30 `up arrow'
    ifTrue: [self position: self position -- (0 @ 1)].
  keyValue = 31 `down arrow'
    ifTrue: [self position: self position + (0 @ 1)].
  keyValue = 29 `right arrow'
    ifTrue: [self position: self position + (1 @ 0)].
  keyValue = 28 `left arrow'
    ifTrue: [self position: self position -- (1 @ 0)]
```

커서 키를 이동하고 morph 를 움직일 수 있는 메서드를 작성했습니다. 마우스가 더이상 대상이 되는 morph의 영역에 없을때, handleKeystroke: 메시지는 전송되지 않기때문에, morph는 키보드입력에 대한 반응을 하지 않게 됩니다. 키보드 입력값을 찾기위해서, Transcript 창에 값을 출력하기 위한 `Transcript show: anEvent keyValue` 내용을 메서드 11.17 에 추가할 수 있습니다. handleKeystroke:의 anEvent 인수는 KeyboardEvent의 인스턴스이며, MorphicEvent의 서브클래스입니다. 키보드 이벤트에 대해 좀 더 알고싶다면 이 클래스를 살펴보시기 바랍니다.

Morphic 애니메이션

Morphic은 2 개의 메인 메서드로 구성되는 간단한 애니메이션 시스템을 제공합니다: stepTime은 각 스텝 사이의 시간간격을 밀레세컨드단위로 지정하고, step[5]은 일정한 간격으로 morph에 전송됩니다. 덧붙여서, startStepping은 스태핑 메커니즘을 on 으로 전환하고, stopStepping은 그 스태핑

[5]stepTime은 실제로 각 스탭 사이의 최소 시간을 의미합니다. 만약 1ms의 stepTime을 요청하면, 스퀵이 너무나 바빠서 사용자의 morph 를 그 정도로 자주 한 단계씩 전환하지 못한다는 사실에 놀라지 마십시오.

메커니즘을 off 상태로 다시 전환합니다. isStepping은 morph가 현재 한 단계씩 진행되는지의 여부를 확인하려 할때 사용합니다.

이 메서드들을 다음과 같이 정의하여 *CrossMorph* 를 깜박이도록 만드십시오.

Method 11.18: 애니메이션 시간 간격 정의하기

```
CrossMorph>>stepTime
  ↑ 100
```

Method 11.19: 애니메이션에서 *step* 만들기

```
CrossMorph>>step
  (self color diff: Color black) < 0.1
    ifTrue: [self color: Color red]
    ifFalse: [self color: self color darker]
```

구현한 애니메이션을 시작하기 위해서, CrossMorph 상의 Inepector 를 열어 할로(할로의 디버그 핸들을 사용) 아래쪽의 작은 Workspace 패널 에 self startStepping 을 입력하고 do it 을 실행합니다. 보조수단을 위해, handleKeystroke: 메서드를 수정하여, + 키와 -키 로 스테핑의 시작와 정지가 가능하게 할 수 있습니다.

다음 코드를 메서드 **??** 에 추가하십시오:

```
    keyValue = $+ asciiValue
        ifTrue: [self startStepping].
    keyValue = $- asciiValue
        ifTrue: [self stopStepping].
```

11.6 대화창 (Interactors)

입력을 위한 프롬프트 상태를 만들기 위해, FillInTheBlank 클래스는 몇 가지 미리 준비된 대화상자를 제공합니다. 예를 들어, `request:initialAnswer:` 메서드는 사용자가 입력한 문자열을 반환합니다. (그림 11.6 참조)

팝업메뉴를 표시하려면, PopupMenu 클래스를 사용합니다:

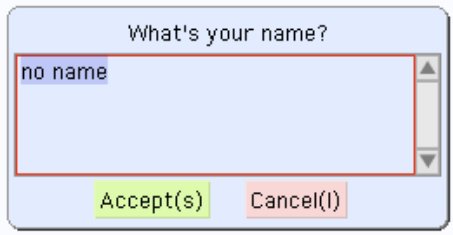


그림 11.6: `FillInTheBlank` request: 'What"s your name?' initialAnswer: `'no name'`이 대화상자를 표시했습니다.

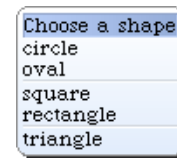


그림 11.7: 팝업 메뉴가 `PopUpMenu>>startUpWithCaption:` 을 표시했습니다.

```
menu := PopUpMenu
    labelArray: #('circle' 'oval' 'square' 'rectangle' 'triangle')
    lines: #(2 4).
menu startUpWithCaption: 'Choose a shape'
```

11.7 드래그 앤 드롭 (Drag-and-drop)

Morphic은 이제까지 배운것 외에도 드래그 앤 드롭을 지원합니다.

수신자*receiver* 역할의 morph와 내려놓아지는*dropped* 역할의 morph, 이 2 가지 객체에 대한 간단한 경우를 생각해 보겠습니다. 수신자는 내려놓은 morph가 주어진 조건을 만족하는 경우에만 내려놓음*drop* 을 수락합니다 : 이 예제에서 morph는 파란색으로 하겠습니다. 수락이 거부되는경우, 내려놓아진 morph가 뭘 할지도 결정하게 됩니다.

먼저 수신자 *morph*를 정의하십시오 :

Class 11.20: 다른 *morph* 를 내려놓을 수 있는 *morph* 를 정의하기

```
Morph subclass: #ReceiverMorph
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'SBE--Morphic'
```

 늘 하던것처럼 *initialization* 메서드를 정의하십시오:

Method 11.21: `ReceiverMorph`의 초기화

```
ReceiverMorph>>initialize
  super initialize.
  color := Color red.
  bounds := 0 @ 0 extent: 200 @ 200
```

만약 받는쪽의 morph는 drop에 대한 수락 또는 거부를 어떻게 판단할까요? 대부분의 경우, 받는쪽과 내려놓아지는쪽의 양쪽 morph 모두 drop 작업에 동의해야 합니다. drop 을 받는쪽은 wantsDroppedMorph:event: 메시지에 반응하는것으로 행동을 결정할 수 있으며, 이 경우 메시지의 첫번짜 인자는 내려놓아진 morph, 두번째 인자는 마우스 이벤트이므로 drop 을 받는객체는 이 상황에 대한 판단을 할 수 있습니다. 예를들어, 받는객체는 drop 시점에서 어떤 modifier key가 눌려져 있는지를 조사할 수 있다는거죠. 내려놓아진 morph, 또는 내려놓아지는 과정에 있는 morph는 수신객체가 drop 을 받는 시점에서 wantToBeDroppedInto: 메시지를 받았을때에 받는객체가 내가 원하는 객체인지 알아볼 수 있는 기회를 가지게 됩니다. wantToBeDroppedInto: 메서드의 기본값(이 값은 Morph 클래스에 정의되어있습니다)으로 true 를 반환합니다.

Method 11.22: *morph*의 색상을 기준하여, 내려놓아진 *morph*를 수락하기

```
ReceiverMorph>>wantsDroppedMorph: aMorph event: anEvent
  ↑ aMorph color = Color blue
```

만약 받는 morph가 떨어진 morph 를 원하지 않는 다면, 내려놓아진 morph에는 어떤일이 일어날까요? 이 경우에 대한 기본동작은 아무것도 하지 않는것으로 되어있으며, 이것은 내려놓아진 morph가 받는 morph 상단에 위치하며 받는 모프와 상호작용은 하지 않는다는것을 의미합니다. 내려놓아진 morph에 대한 좀 더 직관적인 동작은 해당되는 morph의 드래그 이전, 원래의 위치로 돌아가는 것입니다. 이렇게 원래위치로 되돌아가는 작업은 수신객체가 내려놓아진 morph 를 원하지 않을 때, 메시지 repelsMorph:event: 에 true 로 답변할때 진행됩니다.

Method 11.23: 내려놓아진 *morph*가 거절 될 때 해당되는 *morph*의 동작을 변경하기

```
ReceiverMorph>>repelsMorph: aMorph event: ev
  ↑ (self wantsDroppedMorph: aMorph event: ev) not
```

여기까지가, 수신객체에 대해 알아야할 모든 내용입니다.

 워크스페이스에서 *ReveiverMorph*와 *EllipseMorph*의 인스턴스를 생성해주세요:

```
ReceiverMorph new openInWorld.
EllipseMorph new openInWorld.
```

수신객체에 노란색의 `EllipseMorph` 를 드래그 앤 드롭합니다. 이 작업은 거부될 것이며 떨어뜨려진 객체는 초기 위치로 다시 이동되겠죠.

 이 동작을 변경 하기 위해, *Inspector* 를 사용하여 타원 *morph*의 색상을 변경하십시오. 파랑 모프는 반드시 `ReceiverMorph` 에 대해서는 수락되어야만 합니다.

DroppedMorph 라는 이름을 가진 Morph 클래스의 특별한 서브클래스를 만들고, 이것을 이용해서 좀 더 많은 실험을 해보도록 하겠습니다:

Class 11.24: *ReceiverMorph*에 드래그 앤 드롭을 할 수 있도록 *morph* 를 정의하겠습니다.

```
Morph subclass: #DroppedMorph
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'SBE--Morphic'
```

Method 11.25: *DroppedMorph* 초기화 하기

```
DroppedMorph>>initialize
  super initialize.
  color := Color blue.
  self position: 250@100
```

떨어지게된 morph가 받는객체에서 거절될때, 해야할 작업을 설정할 수 있습니다. 여기서, morph는 마우스 포인터에 달려있게 될겁니다"

Method 11.26: 모프를 내려다 놓았지만, 거절되지 않았다면 반응하기

```
DroppedMorph>>rejectDropMorphEvent: anEvent
  |h|
  h := anEvent hand.
  WorldState
    addDeferredUIMessage: [h grabMorph: self].
  anEvent wasHandled: true
```

이벤트에 hand 메세지를 보내면, HangMorph의 인스턴스인 hand 를 반환하며, 이 객체는 마우스 포인터를 나타내고있고, 마우스 포인터에 대한건 뭐든지 가지고 있습니다. 여기서 drop이 거부된 morph 인 self 를 잡고있어야 한다고 world에게 지시하고 있죠.

 *DroppedMorph*의 인스턴스들을 만들고, 수신자 위에 해당 인스턴스들을 끌어다 놓으십시오.

```
ReceiverMorph new openInWorld.
(DroppedMorph new color: Color blue) openInWorld.
(DroppedMorph new color: Color green) openInWorld.
```

녹색의 morph는 거절되었으므로, 마우스 포인터에 잡혀있게 됩니다..

11.8 완전한 예제

주사위(die)[6]를 굴리기 위한 morph 를 만들어보겠습니다 morph 를 클릭하면, 빠른 반복을 통해 모든면이 변경되어 표시되며, 다시 클릭을 하면 애니메이션을 멈추는 동작이 되도록 할것입니다.

 `Morph` 대신에 `BorderedMorph`의 서브클래스로 *die* 를 정의하도록 하겠습니다.. 왜나하면 여기서 테두리를 활용할 것이기 때문입니다.

Class 11.27: 주사위 *morph* 정의하기

```
BorderedMorph subclass: #DieMorph
  instanceVariableNames: 'faces dieValue isStopped'
```

[6] NB: One die, two dice

그림 11.8: Morphic에서의 주사위

```
  classVariableNames: ''
  poolDictionaries: ''
  category: 'SBE--Morphic'
```

인스턴스 변수 faces는 주사위에 대한 면의 개수를 기록하며, 여기서는 주사위의 면들을 9 개까지 지정하도록 하겠습니다.

dieValue는 현재 표시되고있는 면의 값을 나타내며, isStopped는 주사위의 애니메이션이 정지된경우 true가 됩니다. 주사위의 인스턴스를 만들기 위해, DieMorph의 class side에 faces: n 메서드를 정의해서, n 개의 면을 가지는 새로운 주사위를 만들어 보겠습니다.

Method 11.28: 선호하는 면의 개수를 가지는 새로운 주사위를 만들기

```
DieMorph class>>faces: aNumber
  ↑ self new faces: aNumber
```

initialize 메서드는 이전과 같은 방식으로 instance side에서 정의합니다. 덧붙이자면 New는 새롭게 만들어진 인스턴스에 initialize를 전송한다는 것을 기억해 주십시오.

Method 11.29: *DieMorph*의 인스턴스 초기화 하기

```
DieMorph>>initialize
  super initialize.
  self extent: 50 @ 50.
  self useGradientFill; borderWidth: 2; useRoundedCorners.
  self setBorderStyle: #complexRaised.
  self fillStyle direction: self extent.
  self color: Color green.
  dieValue := 1.
```

```
  faces := 6.
  isStopped := false
```

주사위(die)의 모양새를 보다 좋게 하게 위해서, BorderedMorph의 메서드를 몇가지 사용하겠습니다: 입체적인 효과를 얻기 위해서 테두리선을 굵게하거나, 모서리를 다듬거나, 보이는면에 그라디언트를 적용한다던가 등의 작업을 합니다. instance side의 faces: 메서드는, 올바른 인자가 주어지고 있는지를 점검하기 위한 내용을 정의합니다.

Method 11.30: 주사위 면의 개수를 설정하기

```
DieMorph>>faces: aNumber
  `Set the number of faces'
  (aNumber isInteger
      and: [aNumber > 0]
      and: [aNumber <= 9])
    ifTrue: [faces := aNumber]
```

주사위를 만들었을 때, 메시지가 전송되는 순서를 리뷰하는 것이 좋습니다. 예를 들어, `DieMorph face:9` 을 평가한다고 가정해 보겠습니다:

1. 클래스 메서드 DieMorph `class>>faces:`는 DieMorph 클래스에 new 메시지를 발송합니다.
2. new 메서드(Behavior 클래스로부터 상속된 DieMorph 클래스)를 이용해서, 새로운 인스턴스가 생성되고 initialize 를 생성된 인스턴스에 발송합니다.
3. DieMorph에서의 initialize 메서드는 faces 를 초기 값 6 으로 설정합니다.
4. DieMorph `class>>new` 메서드를 실행하면 새로운 인스턴스를 만든후, 새로 만들어진 인스턴스에 클래스 메서드인 DieMorph `class>>faces:`에 `faces: 9`메시지를 발송합니다.
5. 인스턴스 메서드인 `DieMorph>>faces:`가 실행되며 인스턴스 변수인 faces 를 9 로 설정합니다.

drawOn: 메서드를 정의하기 전에, 표시되는 면에 그리는 눈의 위치를 요구하는 메서드를 몇개 만들어보겠습니다.

Method 11.31: *die*의 *faces*에 점 그림을 배치하기 위한 *9* 개의 메서드

```
DieMorph>>face1
  ↑{0.5@0.5}
DieMorph>>face2
  ↑{0.25@0.25 . 0.75@0.75}
DieMorph>>face3
  ↑{0.25@0.25 . 0.75@0.75 . 0.5@0.5}
DieMorph>>face4
  ↑{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75}
DieMorph>>face5
  ↑{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.5@0.5}
DieMorph>>face6
  ↑{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 .
    0.75@0.5}
DieMorph>>face7
  ↑{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 .
    0.75@0.5 . 0.5@0.5}
DieMorph>>face8
  ↑{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 .
    0.75@0.5 . 0.5@0.5 . 0.5@0.25}
DieMorph>>face9
  ↑{0.25@0.25 . 0.75@0.25 . 0.75@0.75 . 0.25@0.75 . 0.25@0.5 .
    0.75@0.5 . 0.5@0.5 . 0.5@0.25 . 0.5@0.75}
```

각 좌표는 1 x 1 크기의 사각형 범위안에 있습니다; 보여지는 면에 표시될 점의 현재 값은 그대로 사용하기에는 너무 작기때문에, 실제 표시작업에 사용하기 위해서는 값을 조절해야 합니다.

drawOn: 메서드는 두 가지 작업을 수행합니다: 주사위의 배경을 그리기 위해서 super 클래스로 메시지를 전송하며, 주사위에 들어갈 점을 그립니다.

Method 11.32: 주사위 *morph* 그리기

```
DieMorph>>drawOn: aCanvas
  super drawOn: aCanvas.
  (self perform: ('face' , dieValue asString) asSymbol)
    do: [:aPoint | self drawDotOn: aCanvas at: aPoint]
```

위 메서드의 뒷부분은 스몰토크의 replection 기능을 사용합니다. 주사위의 면에 점을 그리는 작업은, faceX 메서드로부터 얻은 좌표의 콜렉션에 대해서,

순서대로 각 좌표에 대한 drawDotOn:at: 메시지를 보내서 진행합니다. 알맞은 faceX 메서드를 호출하기 위해, perform: 메서드를 사용해서, ('face', dieValueasString) asSymbol에 의해 만들어진 string 을 메시지로 전송합니다. 이런식의 perform: 메서드에 대한 사용법은 몇번 더 보게 될겁니다.

Method 11.33: 면에 한개의 점 그리기

```
DieMorph>>drawDotOn: aCanvas at: aPoint
  aCanvas
    fillOval: (Rectangle
      center: self position + (self extent * aPoint)
      extent: self extent / 6)
    color: Color black
```

좌표의 숫자범위는 현재 [0:1] 사이에 있기때문에, 주사위의 크기에 맞게 점의 좌표를 계산하기 위해서 self extent * aPoint 를 사용해서 좌표값을 변경하도록 하겠습니다.

 워크스페이스를 통해서 주사위 인스턴스를 만들 수 있습니다.:

```
(DieMorph faces: 6) openInWorld.
```

주사위의 보여지는 면을 바꾸기 위해서, myDie dieValue: 4 와 같은 동작이 가능한 접근자*accessor* 를 만들어보겠습니다.

Method 11.34: 주사위*(die)*의 현재 값을 설정하기

```
DieMorph>>dieValue: aNumber
  (aNumber isInteger
      and: [aNumber > 0]
      and: [aNumber <= faces])
    ifTrue:
      [dieValue := aNumber.
      self changed]
```

이제 주사위의 모든 면을 빠르게 보여주기 위한 애니메이션 시스템을 사용할 것입니다:

Method 11.35: 주사위에 애니메이션 효과 주기

```
DieMorph>>stepTime
```

```
  ↑ 100

DieMorph>>step
  isStopped ifFalse: [self dieValue: (1 to: faces) atRandom]
```

이제 주사위가 작동합니다!

클릭으로 애니메이션을 시작하거나 멈추기 위해, 이전에 마우스 이벤트에 관해 배운 내용을 적용해 보겠습니다. 먼저, 마우스 이벤트의 수신을 활성화합니다:

Method 11.36: 애니메이션을 시작하고 멈추기 위해 마우스 클릭 처리하기

```
DieMorph>>handlesMouseDown: anEvent
  ↑ true

DieMorph>>mouseDown: anEvent
  anEvent redButtonPressed
    ifTrue: [isStopped := isStopped not]
```

이제 주사위를 클릭하면, 주사위가 돌아갑니다.

11.9 캔버스에 대한 더 자세한 내용

drawOn: 메서드는 인자를 한가지만 취급하며, 이 인자로서 요구되는것은 Canvas 클래스의 인스턴스입니다; 여기서 Canvas 란 morph 자신을 draw 하는 영역을 의미합니다. Canvas의 graphics 메서드를 사용하면, 사용자의 morph에 자유로운 다양한 모양을 부여할 수 있습니다. Canvas 클래스의 계층구조를 살펴보면, 이런 변형된 클래스를 얼마든지 찾을 수 있습니다. 일반적으로 사용되는 Canvas 클래스의 변형은 FormCanvas 클래스입니다; Canvas 클래스와 FormCanvas 클래스에서 변형에 사용할 중요한 메서드중 대부분을 찾아낼 수 있습니다. 이러한 메서드에는 points, lines, polygons, rectangles, ellipses, text, 그리고 images 등의 그리기와, rotate, scaling 등이 있습니다.

투명한 morph 또는, 그 외의 그리기 메서드, antialiasing 등을 사용하기 위해서, 다른 종류의 canvas 도 사용할 수 있습니다. 이런것들을 사용해보

고 싶다면 AlphaBelndingCanvas 클래스나 BalloonCanvas 클래스등이 필요할겁니다. 하지만 drawOn: 메서드의 인수로서 FormCanvas의 인스턴스를 받는경우, drawOn: 메서드의 안쪽에서는 어떤 처리를 해야할까요? 다행히도, 어느 canvas라고해도 다른종류의 canvas 로 변환은 가능합니다.

 `DieMorph`에서 *0.5* 값의 투명도*alpha-transparency* 로 *canvas* 를 사용하려면, 다음과 같이 `drawOn:`을 재정의 하시기 바랍니다.

Method 11.37: 반투명한 주사위 그리기

```
DieMorph>>drawOn: aCanvas
  | theCanvas |
  theCanvas := aCanvas asAlphaBlendingCanvas: 0.5.
  super drawOn: theCanvas.
  (self perform: ('face' , dieValue asString) asSymbol)
    do: [:aPoint | self drawDotOn: theCanvas at: aPoint]
```

알아야할건 전부 설명한듯 하군요.

그림 11.9: 알파 투명도로 표시한 주사위

만약 궁금증이 생긴다면, `asAlphaBlendingCanvas:` 메서드를 살펴보시기 바랍니다. 메서드 11.38 에서는, 주사위의 그리기 메서드에서 BallonCanvas 와 antialiasing 을 사용하는 방법을 알수 있습니다.

Method 11.38: *AntiAliasing* 효과를 준 주사위 그리기

```
DieMorph>>drawOn: aCanvas
  | theCanvas |
  theCanvas := aCanvas asBalloonCanvas aaLevel: 3.
  super drawOn: aCanvas.
  (self perform: ('face' , dieValue asString) asSymbol)
    do: [:aPoint | self drawDotOn: theCanvas at: aPoint]
```

```
DieMorph>>drawDotOn: aCanvas at: aPoint
  aCanvas
    drawOval: (Rectangle
      center: self position + (self extent * aPoint)
      extent: self extent / 6)
    color: Color black
    borderWidth: 0
    borderColor: Color transparent
```

11.10 11장 요약

Morphic은 그래픽 인터페이스 구성요소에 있는 그래픽 프레임워크이며, 동적으로 구성이 가능합니다.

- asMorph openInWorld 메시지를 객체에 보내면 morph 로 변환후, 화면에 표시할 수 있습니다.
- 파랑클릭 한후, 나타나는 핸들(각 핸들의 역할을 알려주는 도움말 풍선이 있습니다)를 사용해서 morph 를 조작할 수 있습니다.
- morph 를 다른 morph에 끼워넣는 작업은 드래그앤드롭 또는 addMorph: 메시지를 보내서 수행할 수 있습니다.
- 이미 있는 morph 클래스의 서브클래스를 만든후, initialize, drawOn: 메서드 및 다른 중요한 메서드를 재정의 할 수 있습니다.
- handlesMouseDown: 또는 handlesMouseOver: 메서드등을 재정의하면, morph 를 마우스나 키보드에 반응해서 제어되게 할 수 있습니다.
- morph 객체의 step 메서드(어떤작업)와 stepTime(ms단위의 step 작업의 간격)을 정의하면, morph의 애니메이션을 만들 수 있습니다.
- PopUpMenu와 FillInTheBlank 등, 다양한 종류의 사전 정의 morph 를 사용해서 유저들과 상호작용에 대한 작업을 할 수 있습니다.

제 III 편

고급 스퀵

제 12 장

클래스와 메타클래스

5장 에서 이미 알아봤다시피, 스몰토크에서 모든것은 객체이며, 객체는 어떤 클래스의 인스턴스가 됩니다. 클래스도 예외는 아닙니다: 클래스는 객체이며 클래스 객체는 다른 클래스의 인스턴스가 됩니다. 이런 객체 모델은, 전체가 간결하고 단순하며 우아한 개념으로 되어있습니다. 그리고 객체모델은 객체지향 프로그래밍의 핵심을 가지고 있습니다, 하지만 다른솔루션(또는 다른 환경)을 쓰던 사람에게 이런 통일성은 혼란스러울 수도 있습니다. 지금 보고계시는 12장 에서는 이런 시스템이 마법*magic* 이나 특별한것이 아니며 그리 복잡하지 않다는것을 알려드리고 싶습니다: 사실 간단한 규칙으로 구성되어 있죠. 이런 규칙들을 따라가다 보면 왜 스몰토크의 시스템이 이렇게 구성되어 있는지를 알 수 있을거라 생각합니다.

12.1 클래스와 메타클래스의 규칙

스몰토크의 객체 모델은 적은수의 개념이 동일하게 적용되며 구성되어 있습니다. 스몰토크의 설계자는 Occam의 면도칼로 불리는 규칙을 사용하고 있습니다: 필요 이상으로 모델을 복잡하게 만드는 것들은 사용하지 않는다는거죠.

5장 장에서 배운 객체 모델의 규칙를 기억해보도록 하겠습니다.

Rule 1. 모든 요소는 객체이다.

Rule 2. 모든 객체는 클래스의 인스턴스 이다.

Rule 3. 모든 클래스는 super 클래스를 갖고 있다.

Rule 4. 모든 것은 메시지 발송에 의해 이루어 진다.

Rule 5. 메서드 검색은 상속 관계를 따른다.

이 장의 도입부분에서 얘기했습니다만, Rule 1의 결과는 클래스도 객체라는 의미이고, Rule 2는 클래스 역시 반드시 어떠한 클래스의 인스턴스라는것을 의미합니다.

클래스의 클래스를 메타클래스 라고 합니다.

메타클래스는 사용자가 클래스를 정의할때마다 스몰토크시스템에서 자동으로 만들어집니다. 평상시에 메타클래스를 신경쓸 필요는 없습니다. 하지만, 시스템브라우저에서 "class side" 를 선택하는경우, 사실은 다른 클래스를 보고있다는것을 생각해주시기 바랍니다[1]. 클래스와 메타클래스는, 메타클래스가 클래스의 인스턴스라는점을 제외하면 각각 다른 클래스인것으로 생각해주세요.

클래스와 메타클래스를 보다 적절하게 설명하기 위해, 5장 에 나온 규칙들을 다음의 추가 규칙들로 확장할 필요가 있습니다.

Rule 6. 모든 클래스는 메타 클래스의 인스턴스이다.

Rule 7. 메타 클래스 계층 은 클래스 계층과 평행관계이다.

Rule 8. 모든 메타클래스는 Class 와 Behavior 를 상속한다.

[1] system browser에서 class side 를 누르면 보이는 부분이 class의 instance 인 metaclass 라는 의미인거 같습니다.

Rule 9. 모든 메타클래스는 Metaclass 의 인스턴스이다.

Rule 10. `Metaclass`의 메타클래스는 `Metaclass`의 인스턴스이다.

이상의 10개의 규칙이 스몰토크의 객체모델을 이루는 근간이 됩니다.

일단 5장 에서 언급한 5개의 규칙을 간단한 예제와 함께 살펴보겠습니다. 그리고 같은 예제를 이용해서 추가된 규칙을 자세히 살펴보겠습니다.

12.2 스몰토크객체 모델에 대한 복습

모든 것이 객체가 되기 때문에, 스몰토크에서는 "파랑" 색상 또한 객체 입니다.

```
Color blue --> Color blue
```

모든 객체 클래스의 인스턴스입니다. "파랑" 색상 의 클래스는 Color 입니다.

```
Color blue class  ⟶  Color
```

흥미로운 사실이 있습니다만, 사용자는 투명도*alpha value* 를 지정하면, TranslucentColor 라는 클래스의 인스턴스를 얻을 수 있습니다.

```
(Color blue alpha: 0.4) class  ⟶  TranslucentColor
```

Morph 를 만들고, Morph에 대한 색상으로서 반투명 객체 자체를 지정할 수 있습니다:

```
EllipseMorph new color: (Color blue alpha: 0.4); openInWorld
```

이렇게하면 그림 12.1 같은 결과가 됩니다.

Rule 3에 의해, 모든 클래스는 super클래스를 가지고 있습니다. TranslucentColor의 super클래스는 Color이며, Color의 super클래스는 Object 입니다:

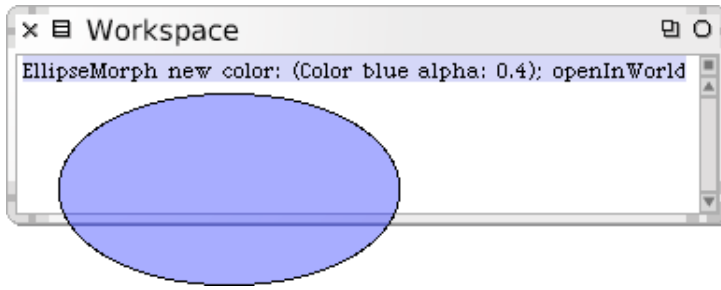


그림 12.1: 반투명 타원

```
TranslucentColor superclass  ⟶    Color
Color superclass             ⟶    Object
```

Rule 4 "모든 동작은 메시지를 보낼 때 일어납니다" 라는 내용을 전제로, blue는 Color 를 대상으로 하는 메시지, class와 alpha:는 "청색" 객체 에 대한 메시지, openWorld는 타원형-Morph에 대한 메시지, 그리고 superclass는 TranslucentColor 및 Color에 대한 메시지가 됩니다. 스몰토크에서 모든것은 객체이기 때문에, 어떤 경우에서도 메시지의 수신자는 객체가 됩니다. 다만 몇몇 경우는 수신자가 클래스인 경우도 있기는 합니다. Rule 5 "메서드 탐색은 상속 관계를 따릅니다" 라는 전제하에, Color blue alpha: 0.4 의 결과로 얻을 수 있는 객체에 class 메시지를 보내면, 그림 그림 12.2 에서처럼, class 메시지는 작동을 위해 결과 자체를 객체로 취급합니다[2].

아래의 그림은 *is-a* 의 상속관계를 보여주고 있습니다. 아래 그림의 내용처럼 "Color blue alpha: 0.4" 의 결과로 통해 얻을 수 있는 translucentColor 객체는, 사실 TranslucentColor 클래스의 인스턴스입니다만, 동시에 Color 객체, 또는 Object의 객체라고 부를수도 있는데, 왜냐하면 translucentColor 객체는 Color 또는 Object에서 선언된 모든 메시지에도 응답이 가능하기 때문입니다. 사실, isKindOf: 메시지를 객체에 보내면 수신자가 인자로 주어진

[2]Color blue alpha: 0.4 의 PrintIt 결과는 (TranslucentColor r: 0.0 g: 0.0 b: 1.0 alpha: 0.4) 가 됩니다. 또 이 결과 자체에 class 메시지를 보내서 PrintIt 하면 결국은 TranslucentColor 라는 결과를 얻을 수 있습니다

클래스와 is-a의 관계에 있는지를 알 수도 있습니다[3].

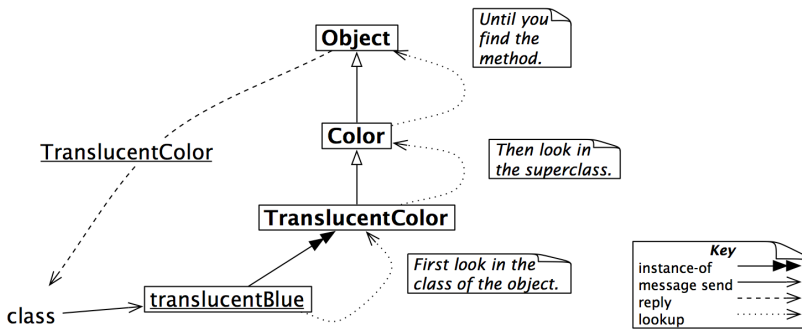


그림 12.2: 메시지를 translucent color에 전송했을때의 상태

12.3 모든 클래스는 메타클래스의 인스턴스이다

12.1절 부분에서 언급했듯이, 인스턴스가 클래스가 되는 클래스는 메타클래스라고 부릅니다[4].

메타클래스는 드러나지 않습니다(묵시적) 메타클래스는 클래스를 정의할 때스몰토크시스템에서 자동으로 생성됩니다. 프로그래머들이 메타클래스에 관해 결코 염려할 필요가 없기 때문에, 그것들을 묵시적이라고 말합니다. 클래스를 작성할때마다, 작성된 클래스를 유일한 인스턴스로 가지는 묵시적 메타클래스가 만들어집니다.

일반적으로 클래스는 global 변수에 의해 참조가 가능합니다만, 메타클래스는 별도의 이름을 가지지 않습니다. 하지만 인스턴스가 되는 클래스를 이용하

[3] isKindOf: 메서드는 Object 클래스에 선언되어있으며 "Answer whether the class, aClass, is a superclass or class of the receiver." 라는 주석을 가지고 있습니다. 이 메서드를 이용하면 수신자의 클래스가 인자로 주어진 클래스에 대해 상위클래스인지 아닌지를 알아낼 수 있습니다. "2 isKindOf: String" 이런정도의 구문을 쓸 수 있다는 의미가 되겠습니다.

[4] 모든 인스턴수화된 클래스는 MetaClass에 소속되어 있다고도 합니다.

면 언제라도 메타클래스를 참조할 수 있습니다. 예를들어, Color의 클래스는 Color class의 인스턴스이며, Object의 클래스는 Object class의 인스턴스입니다.

```
Color class      ⟶     Color class
Object class     ⟶     Object class
```

그림 12.3 은 각 클래스*class* 는 이름을 가지지않는 메타클래스의 인스턴스라는걸 알려주고 있습니다.

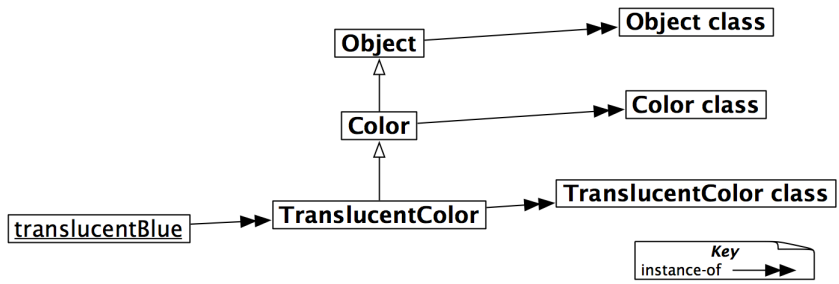


그림 12.3: TranslucentColor 메타클래스와 그것의 super클래스

클래스 또한 객체이며, 메시지를 보내고 정보를 얻어내는일도 간단합니다. 아래 부분을 보도록 하겠습니다:

```
Color subclasses                  ⟶  {TranslucentColor}
TranslucentColor subclasses       ⟶  #()
TranslucentColor allSuperclasses  ⟶  an OrderedCollection(Color
    ObjectProtoObject)
TranslucentColor instVarNames     ⟶  #('alpha')
TranslucentColor allInstVarNames  ⟶  #('rgb' 'cachedDepth'
    'cachedBitPattern' 'alpha')
TranslucentColor selectors        ⟶  an IdentitySet(#alpha:
    #asNontranslucentColor #privateAlpha #pixelValueForDepth:
    #isOpaque
    #isTranslucentColor #storeOn: #pixelWordForDepth:
    #scaledPixelValue32 #alpha
    #bitPatternForDepth: #hash #convertToCurrentVersion:refStream:
    #isTransparent #isTranslucent #setRgb:alpha:
    #balancedPatternForDepth:
```

```
    #storeArrayValuesOn:)
```

12.4 메타클래스 계층은 클래스 계층과 평행관계이다

Rule 7에서는 메타클래스의 super클래스는 임의의 클래스가 될 수 없다고 하고 있습니다: 메타클래스의 super클래스는, 메타클래스의 고유인스턴스의 super클래스에 해당하는 메타클래스만 가능합니다[5].

```
TranslucentColor class superclass     ⟶     Color class
TranslucentColor superclass class     ⟶     Color class
```

위의 결과가 바로 메타클래스의 계층은 클래스 계층과 평행구조를 이룬다는 의미이며, 그림 12.4 는 TranslucentColor에 대한 계층이 어떻게 평생성을 가지는지를 보여줍니다:

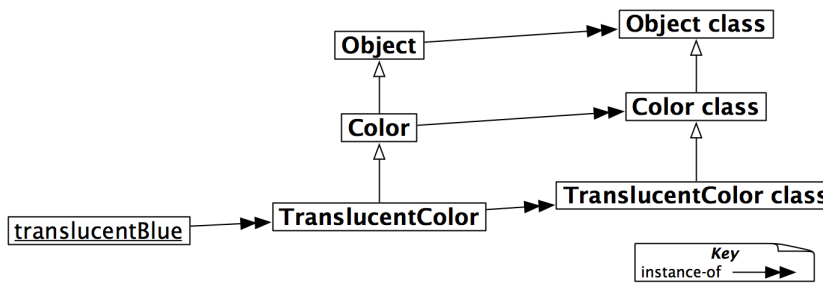


그림 12.4: 메타클래스 계층은 클래스 계층과 평행관계를 이룬다.

[5]TranslucentColor instance(metaclass) 의 superclass > Color 클래스의 instance가 됩니다. 고로 이런 공식이 될 수 있겠습니다. metaclass(instance) superclass>originalclass(class)>superclass of originalclass(class)>metaclass of superclass of originalclass(instance). 결국 메타클래스의 superclass는 원본이 되는 클래스의 상위클래스의 메타클래스가 된다는 얘깁니다. 메타클래스의 superclass는 상위구현의 메타클래스만 된다는 의미가 되겠습니다.

```
TranslucentColor class      ⟶     TranslucentColor class
TranslucentColor class superclass      ⟶     Color class
TranslucentColor class superclass superclass      ⟶     Object
    class
```

클래스와 객체의 일관성 잠시 쉬어가며 생각해보면, 객체 또는 클래스에 메시지를 보내는작업에서 양쪽의 차이가 없다는건 흥미로운 일입니다. 두 경우 모두, 사용할 메서드의 탐색은 수신자클래스부터 시작되서 상속관계를 쫓아갑니다.

따라서 클래스에 보낸 메시지의 탐색은, 메타클래스의 상속계층을 쫓아가게 됩니다. 예를들어, blue 메서드는 Color의 class side에 정의되어 있습니다[6]. TranslucentColor에 blue 라는 메시지를 전송하면 다른 메시지를 찾는것과 동일한 방식으로 메서드를 검색하게 됩니다. 즉 탐색은 TranslucentColor class에서 일단 시작해서 대응되는 메서드를 발견할 때까지 메타클래스 계층을 따라 Color class 까지 더듬어 가게 됩니다(그림 **??** 참고)[7].

```
TranslucentColor blue      ⟶     Color blue
```

TranslucentColor의 인스턴스가 아니라, 일반적으로 Color blue 객체를 얻게된다는걸 주의해주시기 바랍니다-이상한부분은 어느곳에도 없습니다. 당연한 결과인거죠!

위에서 확인한 바와같이, 스몰토크에서는 한결같은 방법으로 메서드 탐색을 하고 있다고 할 수 있습니다. 클래스도 단순히 객체이며, 모든 다른 객체도 동일하게 동작합니다. 클래스를 이용해서 새로운 인스턴스를 만드는것이 가능합니다만, 이런 동작은 클래스가 우현해 new 메시지에 반응할 수 있고, new

[6] java 또는 c++ 에서는 static method에 해당됩니다.

[7] 이건 좀 흥미롭습니다. class의 definition 만 쫓아가는게 아니라 metaclass의 상속계층을 쫓아가는게 기본이군요. 하긴 system browser에서 확인되는게 metaclass 라면 당연한 결과같기는 합니다.

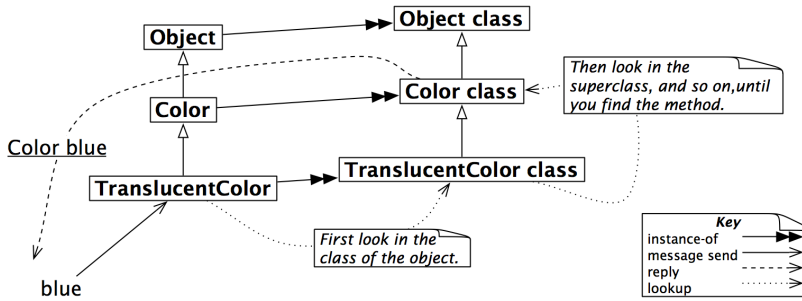


그림 12.5: 클래스를 위한 메시지 탐색은 일반 객체에 대해서도 동일합니다.

에 대한 메서드가 새 인스턴스 생성이 가능하기 때문인것 외에 다른 이유는 없습니다. 일반적으로 클래스 이외의 객체는 new 메시지에 응답할 수 없습니다만, 이렇게 new 메시지의 응답에 대한 동작제한을 한 이유는 new 메서드를 메타클래스 외에도 어디든 추가할 수 있게 되어버리기 때문이죠.

클래스도 객체이기 때문에, inspect가 가능합니다.

 *Color blue*와 *Color* 를 *inspect* 하십시오.

Color의 인스턴스, 그리고 Color class에 대한 조사를 할때는 주의해야합니다. 아마도 헷갈릴 수 있는데, inspector의 제목표시줄은 inspect의 대상이 되는 객체의 클래스 이름을 표시하기 때문이죠.

Color(객체 또는 인스턴스)의 Inspector는 그림 12.6 에서 보이는것처럼, Color 클래스의 super클래스, 인스턴스변수, 메서드 Dictionary 등을 볼 수 있게 해줍니다.

12.5 모든 메타클래스는 Class와 Behavior 를 상속한다.

모든 메타클래스는 클래스이므로, Class 로 부터 상속됩니다. Class의 super 클래스는, ClassDescription이고 그 위의 super클래스는 Behavior 입니다. 스몰토크에서는 모든것이 객체이기 때문에 이 클래스들은 결국은 모두

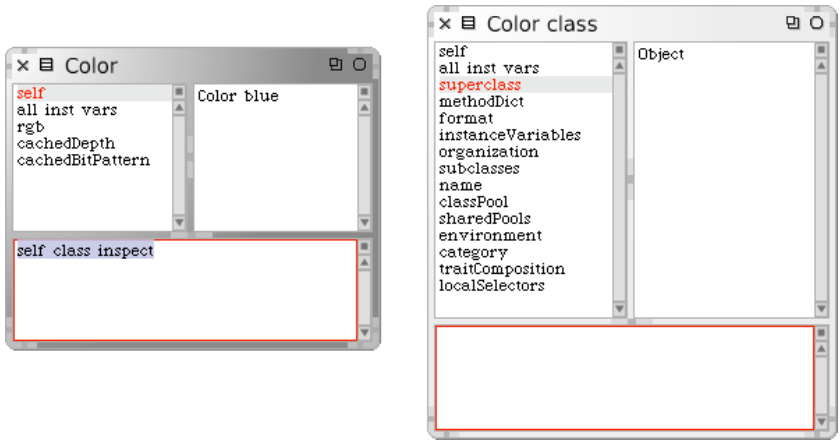


그림 12.6: 클래스 또한 객체 입니다.

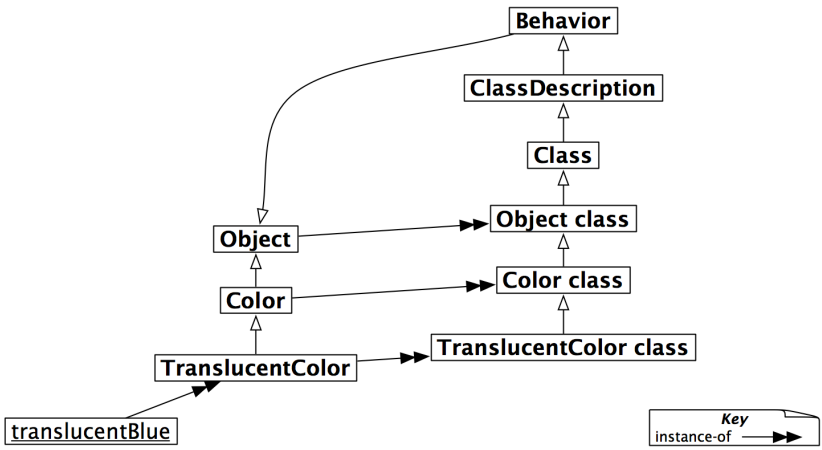


그림 12.7: 메타클래스는 Class와 Behavior로부터 상속됩니다.

Object 부터 상속된것이 됩니다. 그림 12.7 에서 완전한 내용을 확인할 수 있습니다.

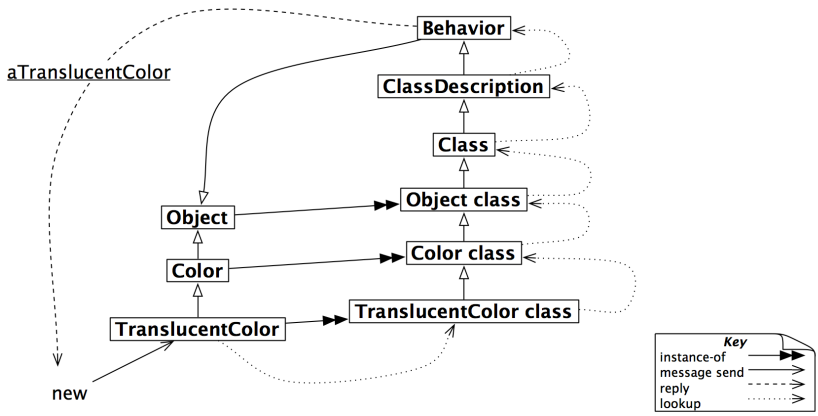


그림 12.8: new 메서드는 일반적인 메세지로 보내졌을때, 메타클래스 관계에서 탐색된다.

`new` 메서드는 어디서 정의되나요? new 메서드는 어디서 정의되고 있으며, 필요한경우 요청에 대해 어떻게 탐색되는지 생각해보는건, 메타클래스가 Class와 Behavior 를 상속하고 있다는점이 중요하다는걸 이해하는데 도움이 됩니다. new 메시지가 클래스에 전송되는 시점에서 그림 12.8 처럼, 메타클래스의 상속관계를 찾고, super클래스인 `Class`, `ClassDescription`, 그리고 `Behavior`의 순서로 메서드를 찾게 되죠.

"어디서 *new* 메서드가 정의되는가" 라는 질문은 매우 중요합니다. new 메서드는 일단 클래스 Behvior에서 정의되며, 필요한 경우, 프로그래머가 정의한 클래스의 메타클래스를 포함한 서브클래스에서 재정의됩니다.

TranslucentColor new가 실행된 결과는 TranslucentColor의 인스턴스가 되며, 비록 메서드가 Behavior에서 발견되었다고 해도 Behavior의 인스턴스가 되는것은 아니라는것에 주의해 주시기 바랍니다. new 메서드는 항상 그 메시지를 받아들여 처리하는 클래스의 self 인스턴스를 반환하며, 다른 클래스에서 new 메서드를 처리한다고 해도 언제나 해당되는 수신자 클래스의 self 인스턴스를 반환하게 됩니다.

```
TranslucentColor new class  ⟶  TranslucentColor
                                "not Behavior!"
```

가장 많이 하는 실수는, 수신자 클래스의 super클래스에서 new 를 찾는 경우입니다. 비슷한 경우로 지정된 크기의 객체를 생성하기 위한 표준 메시지인, new: 도 마찬가지입니다. 예를들면, Array new: 4 는, 4 개의 요소를 가지는 배열을 생성하게 됩니다. 하지만 Array 또는 Array의 super클래스에서 이 메서드의 정의를 찾을 수는 없습니다. 찾고싶다면, Array class와 Array class의 super클래스에서 메서드를 찾아봐야 하며, 왜 이렇게 해야하는가 하면 그곳이 바로 메서드의 검색이 시작되는 곳이기 때문입니다.

Behavior, ClassDescription 과 class의 책임. Behavior는 인스턴스를 가지는 객체를 위한 최소한의 상태*state* 를 제공합니다 : 이 "최소한" 에는 super클래스에 대한 link, 메서드 Dictionary, 인스턴스의 주석 (ex: representation and number) 등이 포함됩니다. Behavior는 Object 를 상속하고 있기때문에, Behavior 자신과 모든 서브클래스는 일반적인 객체처럼 동작할 수 있습니다.

Behavior는 컴파일러에 대한 기본적인 인터페이스도 제공합니다. 메서드 Dictionary의 생성, 메서드의 컴파일링, 인스턴스의 생성 (예를 들어, `new`, `basicNew`, `new:`, 그리고 `basicNew:`), 클래스 계층의 조작 (예를 들어, `superclass:`, `addSubclass:`), 메서드에 대한 접근 (예를 들어, `selectors`, allSelectors, `compiledMethodAt:`), 인스턴스 및 변수들에 대한 접근 (예를 들어, `allInstances`, `instVarNames`), 클래스에대한 접근 (예를 들어, `superclass`, `subclasses`) 그리고 조회를 하기 위한 (예를 들어, `hasMethods`, `includesSelector`, `canUnderstand:`, `inheritsFrom:`, `isVariable`) 메서드들을 제공합니다.

ClassDescription 은 추상클래스이며, Class와 Metaclass 등의 직접적인 서브클래스가 필요로하는 기능을 제공하고 있습니다. ClassDescription은 Behavior가 제공하는 기본에 덧붙여 다음과같은 기능을 제공합니다 : 인스턴스 변수의 이름관리, 프로토콜에 의한 메서드 분류, Change set의 유지 및

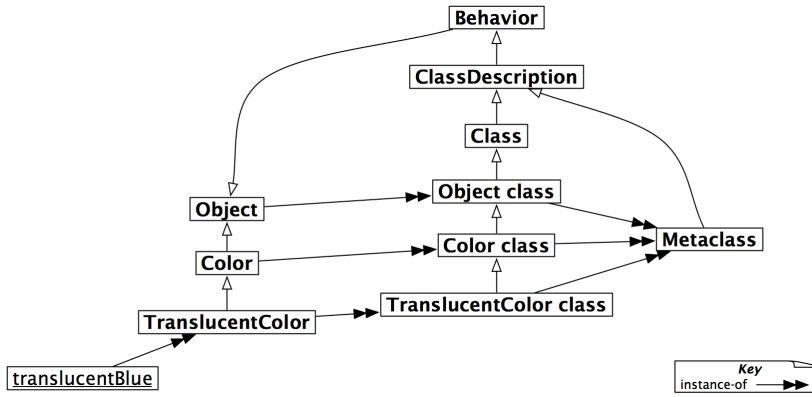


그림 12.9: 모든 메타클래스는 Metaclass 입니다

Change log 관리, 이름에 대한 개념 (추상화*abstract*), 변경부분에 대한 파일출력에 필요한 대부분의 기능 등입니다.

Class는 모든 클래스에서 공통적으로 필요한 동작에 해당됩니다. 클래스 이름, 메서드의 컴파일, 메서드의 보관, 인스턴스변수등을 제공하죠. 그 외에도 클래스의 변수명과 공유 pool 변수 (addClasVarName:, addSharedPool:, initialize) 를 위한 실제 인터페이스도 제공합니다. 메타클래스는 클래스에 대한 유일한 인스턴스이며, 클래스는 인스턴스에 서비스를 제공하기 때문에, 모든 메타클래스는 결국 Class 를 상속받게 됩니다.

12.6 모든 메타클래스는 Metaclass의 인스턴스이다.

그림 12.9 에 보이는 것 처럼, 메타클래스 역시 객체이며, 메타클래스는 Metaclass의 인스턴스 입니다. Metaclass의 인스턴스는 익명의 메타클래스가 되며, 익명의 메타클래스는 각각 유일한 인스턴스를 가지게 됩니다. 이런 각각의 유일한 인스턴스를 클래스라고 합니다.

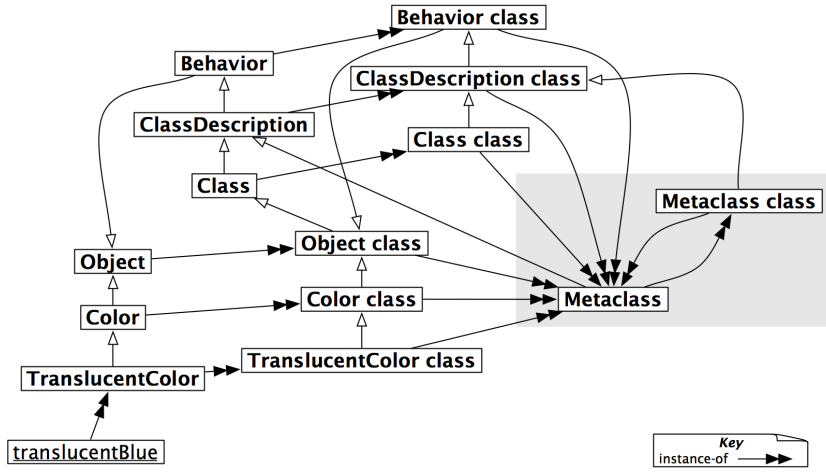


그림 12.10: 모든 메타클래스는 Metaclass의 인스턴스이며 Metaclass의 메타클래스도 마찬가지입니다.

Metaclass는 메타클래스에서 공통적으로 필요한 동작을 제공합니다. 메타클래스의 유일한 인스턴스에 대해서 초기화된 인스턴스를 생성하는 인스턴스 생성 (subclassOf:), 클래스 변수의 초기화, 메타클래스의 인스턴스, 메서드 컴파일, 그리고 클래스의 정보 (상속관계, 인스턴스변수) 에 대한 메서드등이 그것입니다.

12.7 Metaclass의 메타클래스는 Metaclass의 인스턴스이다.

답변이 필요한 질문은 이제 하나가 남았군요. 바로 "Metaclass class 의 클래스는 무엇인가?" 입니다.

답변은 간단합니다. 바로 메타클래스입니다. 물론 시스템내의 다른 메타클래스처럼, Metaclass의 인스턴스여야 합니다.(그림 12.10 참조)

위의 그림은, 모든 메타클래스는 `Metaclass`의 인스턴스이며 `Metaclass`의 메타클래스 또한 동일하게 Metaclass의 인스턴스임을 알려주고 있습니다. 그림 12.9 와 그림 12.10 을 비교하면 `Object class`까지 메타클래스의 계층은 클래스 계층에 대한 완벽한 반영이 된다는걸 알 수 있습니다.

아래의 예제는, 클래스 계층의 조회에 대해서 그림 12.10 이 나타내는 내용을 정확히 보여주고 있습니다.(사실대로 말하자면, 실제로 사용할때는 약간의 차이가 습니다. Object class superclass의 결과는 ProtoObject class가 됩니다, Class가 아니죠. 스퀵에서 Class 까지 가기 위해서는 한단계의 super 클래스를 더 거쳐야 합니다.)

Example 12.1: 클래스 계층

```
TranslucentColor superclass  ⟶  Color
Color superclass             ⟶  Object
```

Example 12.2: 평행 메타클래스 계층

```
TranslucentColor class superclass   ⟶  Color class
Color class superclass              ⟶  Object class
Object class superclass superclass  ⟶  Class "NB: skip
    ProtoObject class"
Class superclass                    ⟶  ClassDescription
ClassDescription superclass         ⟶  Behavior
Behavior superclass                 ⟶  Object
```

Example 12.3: 메타 클래스의 인스턴스

```
TranslucentColor class class  ⟶  Metaclass
Color class class             ⟶  Metaclass
Object class class            ⟶  Metaclass
Behavior class class          ⟶  Metaclass
```

Example 12.4: *Metaclass class*는 *Metaclass*가 됩니다

```
Metaclass class class  ⟶  Metaclass
Metaclass superclass   ⟶  ClassDescription
```

12.8 12장 요약

여러분은 클래스가 어떻게 조직되고있으며, 일관성있는 객체모델이 어떤 영향을 가져오는가에 대해서 좀 더 잘 이해하실 수 있었을거라 생각합니다. 혹시 배운내용을 잊어버리거나 헷갈리는 경우가 발생한다면, 메시지의 전달은 수신자의 클래스에서 원하는 메서드를 찾는경우 가장 중요한 방법이 된다는 사실을 항상 기억해주시기 바랍니다. 메시지의 수신자 클래스에서 메서드를 찾는것이 가장 좋습니다. 만약 메서드를 수신자의 클래스에서 찾을 수 없다면 super클래스에서 조회를 하는것이 좋습니다.

- 모든 클래스는 메타클래스의 인스턴스입니다. `Metaclass`들은 암묵적입니다. 메타클래스는 클래스를 정의할때 그 클래스를 유일한 인스턴스로 하도록 자동으로 만들어집니다.
- 메타클래스 계층은 클래스 계층과 평행적 관계가 됩니다. 클래스의 메서드 검색은 일반적인 객체의 메서드 검색과 병행되며, 메타클래스의 상속관계에 의해 진행됩니다.
- 모든 메타클래스는 `Class`와 `Behavior`로부터 상속을 받습니다. 모든 클래스는 클래스 (Class) 입니다[8]. 메타클래스 또한 클래스이기 때문에, Class 를 상속받고 있습니다. Behavior는 인스턴스들을 가진 모든 존재에게 공통적인 작동을 제공합니다.
- 모든 메타클래스는 `Metaclass`의 인스턴스입니다. `ClassDescription`은 `Class`와 `Metaclsss`에서 필요한 공통것들을 제공합니다.
- Metaclass의 메타클래스는 Metaclass의 인스턴스가 됩니다. 클래스-인스턴스 간의 관계*instance-of* 는 닫힌 루프*closed loop* 를 구성하기 때문에, `Metaclass class class --> Metaclass`의 결과가 됩니다.

[8] Pharo by Example 일본어판에서는 모든 클래스는 Class 의 인스턴스입니다. 라고 되어있습니다

제 IV 편

부록

제 12 장

빈번한 질문과 답변

A 시작하기

FAQ 1. 가장 마지막버전의 *Squeak* 은 어디서 받을 수 있나요?

Answer http://ftp.squeak.org/current_development

FAQ 2. 이 책을 보면서 나는 어떤 *squeak image* 를 사용해야 하지요?

Answer 우리는 Squeak by Example 웹페이지에서 배포하는 이미지를 사용하시길 권장합니다. SqueakByExample.org 홈페이지를 방문하면 옆쪽의 사이드바에서 "Download Squeak" 을 발견할 수 있습니다. 물론 당신은 다른 이미지를 사용해도 됩니다. 다만 책의 내용을 따라 실습하는 동안 몇가지 다른 놀라운 방법을 발견할 수 있습니다.[9]

[9] 이미지가 틀리면 예제들을 진행하는 방법도 달라질 수 있다는 의미입니다

B 컬렉션

FAQ 3. 어떻게 OrderedCollection 을 정렬하나요?

Answer `asSortedCollection` 에 다음 메세지를 보냅니다

```
#(7 2 6 1) asSortedCollection  ⟶  a SortedCollection(1 2 6 7)
```

FAQ 4. 어떻게하면 문자 컬렉션을 *스트링* 으로 변환할 수 있나요?

Answer

```
String streamContents: [:str | str nextPutAll: 'hello' asSet]
     ⟶  'hleo'
```

C 시스템 탐색

FAQ 5. 책에서 자주 언급된 브라우저를 볼 수가 없어요. 어떻게된거죠?

Answer 당신은 아마 OmniBrowser 가 기본으로 설치된 이미지를 사용할 수 있을겁니다. 시스템 브라우저창의 상단-왼쪽 코너에 있는 작은 메뉴 아이콘을 클릭해서 기본값을 변경할 수 있습니다, "X" 와 시스템브라우저 라벨 사이에 타나나는 메뉴에서 "Choose new default browser" 를 선택합니다. `#Browser` 를 선택하면 구형의 기본브라우저를 사용할 수 있습니다. (World ▷ open⋯ ▷ system browser 메뉴에서 환경설정의 변경을 적용할 수 있습니다. 하지만 도구플랩의 브라우저 아이콘으로 드래그한다면 아무일도 일어나지 않습니다.)

FAQ 6. 어떻게 클래스를 검색하나요?

Answer CMD-b (browse) 를 눌러서 클래스 이름으로 하면되고, 또는 CMD-f 를 눌러서 클래스 브라우저의 카테고리 창에서 할 수 있습니다.

FAQ 7. 어떻게하면 *super*클래스에 대한 모든 전송을 찾기/검색 할 수 있습니까?

Answer 두번째 방법이 훨씬 빠릅니다:

```
SystemNavigation default browseMethodsWithSourceString: 'super'.
SystemNavigation default browseAllSelect: [:method | method
    sendsToSuper ].
```

FAQ 8. 어떻게 하면 계층구조에 속한 모든 super 클래스에 대한 전송을 탐색할 수 있습니까?

Answer

```
browseSuperSends := [:aClass | SystemNavigation default
    browseMessageList: (aClass withAllSubclasses gather: [ :each |
        (each methodDict associations
            select: [ :assoc | assoc value sendsToSuper ])
                collect: [ :assoc | MethodReference class: each
    selector: assoc key ] ])
    name: 'Supersends of ' , aClass name , ' and its subclasses'].
browseSuperSends value: OrderedCollection.
```

FAQ 9. 어떻게하면 클래스에 의해 새로 도입된 메서드를 찾을 수 있나요? *(*오버라이드된 메서드는 포함하지 않습니다*)*

Answer 이 아래에서 True 를 이용해서 의해 도입된 것을 찾는 방법을 알려드립니다:

```
newMethods := [:aClass| aClass methodDict keys select:
    [:aMethod | (aClass superclass canUnderstand: aMethod) not ]].
newMethods value: True  ⟶  an IdentitySet(#asBit)
```

FAQ 10. 어떻게 *(*구현*)*하면 어떤 클래스의 메서드는 추상적이라고 말할 수 있나요?

Answer

```
abstractMethods :=
    [:aClass | aClass methodDict keys select:
        [:aMethod | (aClass>>aMethod) isAbstract ]].
abstractMethods value: Collection  ⟶  an
    IdentitySet(#remove:ifAbsent: #add: #do:)
```

FAQ 11. 나는 *AST* 표현식의 *view* 를 어떻게 생성해야 하나요?

Answer squeaksource.com에서 AST를 로드한 이후, 다음을 실행합니다:

```
(RBParser parseExpression: '3+4') explore
```

FAQ 12. 시스템 안의 모든 특성*Traits* 을 찾으려면 어떻게 해야하나요?

Answer

```
Smalltalk allTraits
```

FAQ 13. 어떻게 하면 클래스에서 사용가능한 특성*Traits* 찾을 수 있나요?

Answer

```
Smalltalk allClasses select: [:each | each hasTraitComposition
    and: [each traitComposition notEmpty ]]
```

D A.4 몬티첼로와 SqueakSource 사용하기

FAQ 14. 어떻게 해야 *Squeaksource* 를 불러들일 수 있나요?

Answer

1. 원하는 프로젝트를 squeaksource.com 에서 찾으세요
2. 등록된미리보기코드*registration code snippet* 를 복사합니다

3. open ▷ Monticello browser 를 선택하세요
4. +Repository ▷ HTTP 를 선택하세요.
5. 등록된미리보기코드*registration code snippet* 를 붙여넣기하고 선택하세요; 당신의 비빌번호를 입력하세요.
6. 새로운 저장소를 선택하고 Open 을 선택합니다.
7. 선택하면 마지막 버전을 불러들이게 됩니다.

FAQ 15. 나는 *SqueakSource project* 를 어떻게 생성해야 하나요?

Answer

1. squeaksource.com 으로 갑니다.
2. 당신을 new member로 등록하세요
3. 프로젝트를 등록합니다 (name = category)
4. 등록된미리보기코드*registration code snippet* 를 복사합니다
5. open ▷ Monticello browser
6. +Package 를 카테고리에 추가합니다
7. 패키지를 선택합니다
8. +Repository ▷ HTTP
9. 등록된미리보기코드*registration code snippet* 를 붙여넣기하고 선택하세요; 당신의 비빌번호를 입력하세요.
10. Save하면 첫번째 버전이 저장됩니다.

FAQ 16. `Number`를 `Number>>chf`로 확장하는데, 어떻게 해야 몬티첼로가 내 `Money` 프로젝트를 몬티첼로의 일부분으로 인식하게 할 수 있는건가요?

Answer 메서드-카테고리의 이름을 `*Money`라고 넣습니다. Monticello는 *package 같은 이름으로 시작되는 카데고리의 메서드를 정리하고 해당되는 패키지에 포함시킵니다.

E A.5 도구

FAQ 17. *SUnit TestRunner* 를 열려면 어떻게 해야 하나요?

Answer 워크스페이스에서 `TestRunner open.` 을 실행하면 됩니다.

FAQ 18. 리팩토링 브라우저는 어디서 찾아야 하나요?

Answer AST를 로드할때 squeaksource.com에서 가져온 리펙토링엔진에서 찾을 수 있습니다.
www.squeaksource.com/AST www.squeaksource.com/RefactoringEngine

FAQ 19. 브라우저를 어떻게 기본값으로 등록하나요?

Answer 브라우저창 좌측상단의 메뉴아이콘을 클릭합니다.

F A.6 정규표현식과 해석

FAQ 20. 어떻게 해야 정규식을 사용할 수 있나요?

Answer Vassili Bykov 의 정규식 패키지를 불러들이면 됩니다. 다음의 주소를 참조해주세요 : www.squeaksource.com/Regex.html

FAQ 21. 정규식패키지에 대한 문서는 어디서 찾아야 하나요?

Answer `theVB-Regex` 카테고리의 RxParser 프로토콜 문서를 보세요.

FAQ 22. 파서를 작성(제작)하기위한 도구가 있나요?

Answer SmaCC 를 사용하세요 — 스몰토크컴파일러의 컴파일러입니다. SmaCC-lr.13 이상의 버전을 설치하세요. www.squeaksource.com/SmaccDevelopment.htm 에서 로드할 수 있습니다. 아주 좋은 온라인설명서가 있습니다:
www.refactory.com/Software/SmaCC/Tutorial.html

FAQ 23. 파서를 작성(제작)하기 위해서 *SqueakSource SmaccDevelopment* 로부터 어떤 패키지를 불러들여야 하나요?

Answer 가장 최근버전의 SmaCCDev 를 로드하세요 — 런타임버전을 이미 가지고 있을겁니다.(스퀵3.8을 위한 SmaCC-Development)

참고 문헌

[1] A. Sharp, *Smalltalk by Example*. McGraw-Hill, 1997.

[2] K. Beck, *Test Driven Development: By Example*. Addison-Wesley, 2003.

[3] S. R. Alpert, K. Brown, and B. Woolf, *The Design Patterns Smalltalk Companion*. Addison Wesley, 1998.

[4] K. Beck, *Smalltalk Best Practice Patterns*. Prentice-Hall, 1997.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison Wesley Professional, 1995.

[6] B. Woolf, "Null object," in *Pattern Languages of Program Design 3* (R. Martin, D. Riehle, and F. Buschmann, eds.), pp. 5–18, Addison Wesley, 1998.

[7] A. Goldberg and D. Robson, *Smalltalk 80: the Language and its Implementation*. Reading, Mass.: Addison Wesley, May 1983.

[8] W. LaLonde and J. Pugh, *Inside Smalltalk: Volume 1*. Prentice Hall, 1990.

찾아보기