GNOME3 Application Development Beginners Guide

GNOME3 Application Development Beginners Guide Korean Edition

□□

□□ 232

GNOME3ApplicationDevelopmentBeginnersGuide:Copy

저작권 정보

저작권 정보

GNOME 3 애플리케이션 개발

초보자 지침서

GNOME 애플리케이션 개발을 이해하기 위한 단계별 실습 지침서

Mohammad Anwari (저자)

첫 출판일: 2013년 2월

생산 참조번호: 1080213

출판회사

Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-84951-942-7

http://www.packtpub.com

Cover Image by Duraid Fatouhi (duraidfatouhi@yahoo.com)

Notes

GNOME3ApplicationDevelopmentBeginnersGuide:Cred

Credits

Credits

Author

Mohammad Anwari

Reviewers

Dhi Aurrahman

Joaquim Rocha

Acquisition Editor

Mary Jasmine

Lead Technical Editor

Ankita Shashi

Technical Editors

Charmaine Pereira

Dominic Pereira

Copy Editors

Laxmi Subramanian

Aditya Nair

Alfida Paiva

Ruta Waghmare

Insiya Morbiwala

Project Coordinator

Abhishek Kori

Proofreader

Mario Cecere

Indexer

Tejal Soni

Graphics

Aditi Gajjar

Production Coordinator

Aparna Bhagat

Cover Work

Aparna Bhagat

Notes

GNOME3ApplicationDevelopmentBeginnersGuide:Abou

저자 정보

저자 정보

Mohammad Anwari 는 소프트웨어 개발에만 13년의 경험을 가진 인도네시아 출신의 소프트웨어 해커다. 그는 Linux 기반 시스템, GNOME의 애플리케이션, Qt 플랫폼으로 작업해왔다. 또한 제한된 장치 및 데스크톱 애플리케이션의 개발부터 고급 트래픽 서버 시스템과 애플리케이션까지 작업해왔다.

Nokia/MeeGo에서 일하기 위해 핀란드로 이사를 하기 전이었던 닷컴(dotcom) 시대에 그는 신생 벤처기업을 시작했다. 지금은 인도네시아로 돌아가 Node.js와 Linux 기반의 프로젝트에 중점을 둔 새로운 벤처기업을 설립하여 기업가로 활동 중이다. 여가 시간에는 인도네시아에서 가장 큰 오픈 소스 프로젝트에 속하는 BlankOn의 이사로 활동한다.

과거에 Linux에 관한 서적을 인도네시아어로 여러 권 출판한 이력이 있다.

> 우리 가족, Rini, Alif, Abil의 엄청난 지지가 없었다면 이 책을 써낼 수 없었을 것이다.

Notes

GNOME3ApplicationDevelopmentBeginnersGuide:Abou

검수자 정보

검수자 정보

Dhi Aurrahman 은 Labtek Indie의 프로젝트 매니저로, 다양한 장치와 플랫폼에 특별히 만들어진 상호작용형 애플리케이션의 개발을 선도한다. 그 전에는 한국에서 삼성 일렉트로닉스와 함께 실시간 컴퓨터 시각 시스템을 기반으로 다양한 프로젝트를 작업했다. C++ 프로그래밍에 숙련된 기술을 겸비하고 있으며, 데스크톱과 모바일을 포함해 다양한 플랫폼에 Qt를 이용한 애플리케이션 개발과 JavaScript 프로그래밍에도 능숙하다. Dhi는 인도네시아 반둥공과대학 계산 물리학에서 이학사 과정(B.Sc)을 마쳤고, 한국 전남대학교 컴퓨터 공학과에서 컴퓨터 시각과 기계에 중점을 두어 공학 석사 학위를 취득하였다.

Joaquim Rocha 는 포르투갈 출신의 소프트웨어 개발자로, 컴퓨터 과학에서 이학 석사를 수료하였다. 그래픽 사용자 인터페이스의 개발에 6년 간 경험을 쌓아왔으며, 여러 무료 소프트웨어 프로젝트에 참여해왔다.

GNOME Foundation의 구성원이자 GNOME 데스크톱용으로 개발된 가장 완전한 OCR 애플리케이션인 OCRFeeder를 작성한 사람이기도 하다.

무료 소프트웨어 자문 단체인 Igalia에서 작업하면서 GNOME 기술로 전부 작성된 세계 첫 무료 소프트웨어 스켈레톤(skeleton) 추적 라이브러리를 생성하였다.

Notes

GNOME3ApplicationDevelopmentBeginnersGuide:Pack

www.PacktPub.com

www.PacktPub.com

지원 파일, eBook, 할인 혜택과 기타 정보

책과 관련된 지원 파일과 다운로드는 http://www.PacktPub.com 을 방문하길 바란다.

Packt 는 발매되는 책마다 PDF, ePub 파일과 함께 eBook 버전을 제공한다는 사실을 알고 있었는가? eBook 버전은 http://www.PacktPub.com 에서 업그레이드가 가능하며, 서적을 구매한 고객은 eBook을 할인 가격으로 구매할 수 있다. 자세한 정보는 service@packtpub.com 으로 연락을 바란다.

http://www.PacktPub.com 에서는 무료 기술 기사를 읽고, 무료 뉴스레터를 구독하여 Packt 서적과 eBooks에 할인 혜택을 받을 수 있다.

IT 질문에 즉각적인 해답이 필요한가? PacktLib은 Packt의 온라인 디지털 서적 라이브러리다. 여기서는 Packt의 모든 서적들을 찾아 읽고 검색할 수 있다.

구독해야 하는 이유

- Packt가 출판한 모든 책을 모두 검색할 수 있다.
- 내용의 복사, 붙이기, 인쇄, 북마킹이 가능하다.
- 언제든지 웹 브라우저를 통해 접근 가능하다.

Packt 회원을 위한 무료 혜택

http://www.PacktPub.com 를 통해 Packt에 계정을 만들었다면 PacktLib로 접근하여 무료 서적 9권을 모두 볼 수 있다. 로긴 인증서를 사용하면 즉시 접근이 가능하다.

Notes

GNOME3ApplicationDevelopmentBeginnersGuide:Prefa

서문

서문

이 서적은 Vala와 Javascript 프로그래밍 언어를 이용한 GNOME3 애플리케이션의 개발에 관한 책이다. 이는 GNOME3 플랫폼에서 GTK+, Clutter, HTML5 애플리케이션을 빌드하도록 안내한다. 데이터 접근, 멀티미디어, 네트워킹, 파일시스템과 같이 GNOME 3에 특정적인 하위시스템을 다룬다. 뿐만 아니라 지역화(localization)나 테스팅과 같은 훌륭한 소프트웨어 공학 실제를 다루기도 한다.

본문에서 다룰 내용

제 1장, "GNOME3와 SDK 설치하기" 에서는 GNOME3와 소프트웨어 개발 키트를 자주 사용되는 Linux 배포판에서 설치하는 것을 논한다.

제 2장, "무기 준비하기" 에서는 본문에 사용된 통합 개발 환경과 사용자 인터페이스 디자이너, 즉 Anjuta와 Glade의 기본적인 사용을 논한다.

제 3장, "프로그래밍 언어" 는 Vala의 기본 내용과 Seed를 이용한 JavaScript 프로그래밍을 다룬다. Vala와 JavaScript에 익숙하지 않은 독자라면 뒤에 실린 장들을 이해하는 데 기반이 되는 내용이다.

제 4장, "GNOME 코어 라이브러리 사용하기" 는 GNOME 코어 라이브러리에서 자주 사용되는 기능을 안내할 것이다.

제 5장, "그래픽 사용자 인터페이스 애플리케이션 빌드하기" 는 GTK+와 Clutter를 이용해 GUI 애플리케이션을 빌드하는 단계를 설명한다.

제 6장, "위젯 생성하기" 는 처음부터 GTK+ 위젯을 생성하는 방법을 설명한다. 이번 장은 위젯의 확장과 맞춤 설정에 관해 논한다.

제 7장, "멀티미디어 즐기기" 에는 GStreamer에 관한 방대한 정보가 포함되어 있다. 멀티미디어 스트림을 다루고 필터를 스트림으로 적용하는 것에 관해 다룬다.

제 8장, "데이터 다루기" 에서는 TreeView API 패밀리로 데이터를 표현하는 것을 설명한다. 데이터 표현을 표시하는 동시 Evolution Data Server로부터 데이터를 얻는 것을 논한다.

*제 9장, "GNOME*을 통해 *HTML5* 애플리케이션 활용하기" 에서는 WebKit를 GTK+ 애플리케이션에 넣는 방법을 설명한다. WebKit에서 실행되는 JavaScript가 Vala로 쓰인 백엔드 시스템과 대화하도록 만들 수 있는 JavaScriptCore 라이브러리에 대해 논하겠다.

제 10장, "데스크톱 통합" 은 GNOME3 데스크톱과 정확히 통합하는 애플리케이션의 생성에 관해 논한다. 또한 D-Bus, 세션 관리, 키링(keyring), 런처, 알림 서비스도 다룬다.

제 11장, "애플리케이션 국제화하기" 에서는 GNOME3 애플리케이션에서 국제화와 지역화를 논한다. 그에 더해 지역화 과정의 제안을 제공하기도 한다.

제 12장, "품질이 좋으면 모든 게 쉬워진다" 는 단위 테스팅(unit testing)과 스터빙(stubbing)에 관해 논한다. GTK+ 애플리케이션과 GUI가 아닌 애플리케이션의 테스트를 다룬다.

제 13장, "흥미로운 프로젝트" 는 웹 브라우저와 Twitter 클라이언트를 빌드하기 위해 두 가지 흥미로운 프로젝트를 제공한다. 앞에 소개한 장들에서 학습한 측면들을 많이 다루고 두 프로젝트에서 사용한다.

무엇이 요구되는가

기본 수준의 객체 지향 프로그래밍 지식을 요구한다. Vala 또는 JavaScript에 대한 사전 지식이 있다면 더 즐거운 경험이 될 것이다. 본문의 내용을 따라 하기 위해서는 Fedora, openSUSE, Ubuntu, 또는 Debian 기반의 Linux 배포판(권장)의 최신 버전을 설치하도록 한다.

누구를 위한 책인가

이 책은 배포 플랫폼으로 GNOME3을 사용하고자 하는 소프트웨어 개발자에게 적합하다. GNOME3 라이브러리 중 다수는 Linux, OSX, Windows에서도 이용할 수 있으므로 다중 플랫폼 애플리케이션을 생성하려는 개발자에게 유용한 책이 되겠다.

규약

여러 개의 제목이 자주 사용될 것이다. 프로시저 또는 작업을 완료하는 방법에 대한 명확한 설명을 제공하기 위해 아래를 이용하겠다.

실행하기 - 제목

1. Action 1
2. Action 2
3. Action 3

지침에는 종종 의미를 풀어놓은 추가 설명이 필요하므로 아래와 같은 내용이 따라온다.

무슨 일이 일어났는가?

이 제목은 방금 완료한 작업이나 지침의 결과물을 설명한다. 본문에서 찾을 수 있는 학습 보조물로는 다음이 포함된다.

팝 퀴즈 - 제목

이해도를 검사할 수 있도록 짧은 선택형 질문이 제공된다.

시도해보기 - 제목

실용적인 과제를 통해 학습한 내용을 실험하도록 개념들을 제공한다.

정보의 유형을 구별하는 여러 스타일의 텍스트를 발견할 것이다. 이러한 스타일의 예제와 그 의미를 설명해 보겠다.

텍스트에 코드의 단어는 "Modify configure.ac to include WebKitGTK+ into the project.(WebKitGTK+가 프로젝트에 포함되도록 configure.ac를 수정하라.)"는 식으로 표시된다. 코드의 블록은 아래와 같이 설정된다.

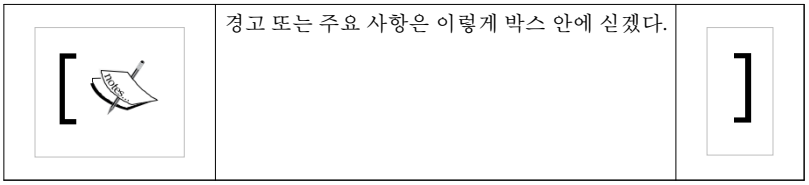
```
using GLib;
using Gtk;
using WebKit;

public class Main : WebView
{
    public Main ()
    {
load_html_string("<h1>Hello</h1>","/");
    }
    static int main (string[] args)
    {
Gtk.init (ref args);
varwebView = new Main ();
```

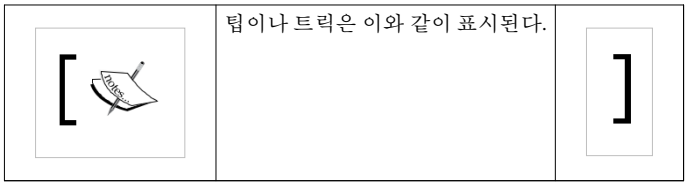
```
var window = new Gtk.Window();
window.add(webView);
window.show_all ();

Gtk.main ();
        return 0;
    }
}
```

모든 명령행 입력이나 출력은 아래와 같이 쓴다.

```
LANGUAGE=id LC_ALL=id_ID.utf8 src/hello-i18n
```

새로운 용어 나 중요한 단어 는 볼드체로 표시한다. 화면에 메뉴 또는 대화상자 박스를 통해 표시되는 단어는 "Click **Continue**, finish the setup of **GTranslator** and go ahead and open the id.po file using its menu.(**Continue** 를 누르고, **GTranslator** 의 셋업을 완료한 후 그 메뉴를 이용해 id.po 파일을 연다.)"와 같이 나타난다.





독자 피드백

독자로부터 피드백은 언제나 환영이다. 책을 읽고 난 후 어느 부분이 좋은지, 어느 부분이 마음에 안 드는지 알려주길 바란다. 독자들의 피드백은 독자들이 최대한으로 활용할 수 있는 제목을 만드는 데 매우 중요하다.

일반적인 피드백은 책 제목을 이메일 제목으로 하고 메시지를 작성하여 feedback@packtpub.com 으로 보내주길 바란다.

전문적으로 학습하고 싶은 주제가 있을 경우, 또는 이 책을 쓰는 데나 기여하는 데 관심이 있다면 http://www.packtpub.com/authors 에서 저자 안내문을 확인하도록 한다.

고객 지원

이제 Packt 서적을 소유했으니 최대한으로 활용하도록 도울 일만 남았다.

예제 코드 다운로드하기

계정을 통해 구입한 Packt 서적에 실린 예제 코드 파일은 모두 http://www.packtpub.com 에서 다운로드할 수 있다. 다른 곳에서 구매했다면 http://www.packtpub.com/support 를 방문하여 등록하면 파일이 본인의 이메일로 직접 전송될 것이다.

오식

내용의 정확도를 보장하기 위해 최대한으로 주의를 기울였지만 실수가 일어나기도 한다. 자사의 출판물에서 텍스트 또는 코드에 실수가 발견될 경우 보고해준다면 매우 감사하겠다. 이를 통해 다른 독자들의 불편함을 덜고, 앞으로 출판될 버전들이 개선되도록 도와줄 것이다. 어떤 실수든 http://www.packtpub.com/submit-errata 를 방문해 책을 선택하고 erratasubmissionform 링크를 클릭하여 상세한 내용을 기입해주길 바란다. 오식이 확인되면 제출 내용이 수용되어 본사의 웹사이트로 업로드되거나, Errata라는 제목으로 된 섹션에 실린 기존의 오식 리스트에 추가될 것이다.

저작권 침해

어떤 매체든 인터넷상에서 저작권 보호 자료의 침해 문제는 계속해서 발생한다. Packt에서는 저작권과 라이센스의 보호를 매우 심각하게 취급한다. 인터넷에서 어떠한 형태로든 자사 제품의 불법 복사본이 발견되면 법적 방책을 마련할 수 있도록 즉시 자료를 사용하는 위치의 주소나 웹사이트명을 알려주길 바란다.

저작권을 침해한 것으로 의심되는 자료의 링크는 copyright@packtpub.com 으로 보내주길 바란다.

저자의 권리, 그리고 독자에게 귀중한 자료를 제공할 수 있는 자사의 능력을 보호하도록 도와주는 독자들에게 감사의 말을 전한다.

질문

책과 관련해 문제가 생길 시 questions@packtpub.com 으로 연락을 준다면 최선을 다해 답하도록 하겠다.

Notes

GNOME3ApplicationDevelopmentBeginnersGuide:Chap 01

제 1 장 GNOME3와 SDK 설치하기

GNOME3와 SDK 설치하기

*GNOME3*는 *1999*년 처음 배포된 이후 가장 훌륭하면서 최신의 *GNOME* 버전이다. *GNOME3*는 기존의 버전들에 비해 데스크톱 사용자의 경험에 돌파구를 제시한다. 유체 애니메이션과 함께 부드러운 사용자 인터페이스를 제공하는 동시 *GNOME* 셸을 새로운 사용자 경험*(UX)*으로 소개함으로써 현대 데스크톱이 어떻게 작동하는지에 대해 남다른 은유법을 도입한다. 하지만 하드웨어 구성이 지원되지 않을 때를 대비해 전통적인 사용자 경험도 고수한다.

GNOME 은 그 단순성으로 잘 알려져 있으며, 많은 유명한 Linux 배포판에서 기본 데스크톱 환경으로 선택되고 있다. 사용자 인터페이스 지침서를 가진 첫 자유 소프트웨어 유형에 해당하기도 하여 유용성과 사용자 친화성을 보장한다.

GNOME은 개발하면서 아키텍처 또한 확장되어 세계에서 자유롭게 이용할 수 있는 최고의 데스크톱에 속한다. 그에 더불어 하드웨어의 사용을 최대화하는 동시 사용자가 컴퓨터를 사용할 때 즐거운 시간을 제공하기 위해 시스템에서 요구되는 첨단 기술은 점점 더 증가하고 있다.

더 나아가기 전에 GNOME과 개발 환경을 컴퓨터에 적절하게 설치해야 한다. 먼저 자주 사용되는 몇 개의 Linux 배포판에서 이들을 설치하는 방법을 이야기하겠다. 이번 장에서 구체적으로 다룰 주제는 다음과 같다.

- 시스템 요구사항 이해하기
- 본 장에서 다루게 될 기본 GNOME3 아키텍처
- 다양한 Linux 배포판에서 GNOME3와 SDK 설치하기

이제 시작해보도록 하자.

시스템 요구사항

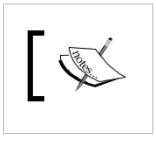
GNOME3는 설치된 하드웨어의 유형을 대상으로 두 개의 사용자 경험**(UX)** 집합, **GNOME Shell**과 **GNOME Panel/Fallback** 을 제공한다. 두 개의 UX는 서로 다른 요구사항을 갖지만 기본 사항은 공유한다.

- 800 MHz CPU power (최적 성능에 최소 1 GHz)
- 램 512 MB (최적 성능에 최소 1 GB)
- 2GB 이상의 이용 가능한 하드 드라이브 공간 (다른 애플리케이션을 설치한다면 그 이상)

GNOME Shell

이는 GNOME3에 가장 최근에 추가된 내용으로, 겉모습이 좋은 사용자 경험을 제공한다. 해당 UX는 시스템에 적절하게 설치된 OpenGL 가능(OPGL-capable) 영상 카드를 통해 3D 가속 기능을 필요로 한다. Intel, API, nvidia 브랜드의 영상 카드는 대부분 문제 없이 작동할 것이다.

하지만 OpenGL 가능한 영상 카드를 갖고 있다 하더라도 영상 카드가 적절한 드라이브로 활성화되도록 확보해야 하는데, 그렇지 못할 경우 시스템은 해당 UX를 사용하지 못하도록 할 것이다.



자신의 영상 카드가 해당 UX와 작동하는지 확인하려면 [1] 에서 **h-node page** 를 방문하여 **works with 3D acceleration** 사항을 확인한다.

GNOME Panel/Fallback

무난하고 오래된 GNOME 데스크톱으로, 좀 더 기본적이고 덜 매력적인 사용자 경험을 제공한다. 시스템은 OpenGL을 활성화하지 못하면 이 UX로 돌아온다. 해당 UX는 기존의 GNOME 버전과 비슷하지만 사용자 인터페이스가 약간 수정되었기 때문에 GNOME 셸 사용자 인터페이스의 상태와 유사하게 보일 것이다. 가령, 메인 UI는 완전히 다르지만 화면 잠금(lock scree)은 두 UX에서 꽤 비슷하다.

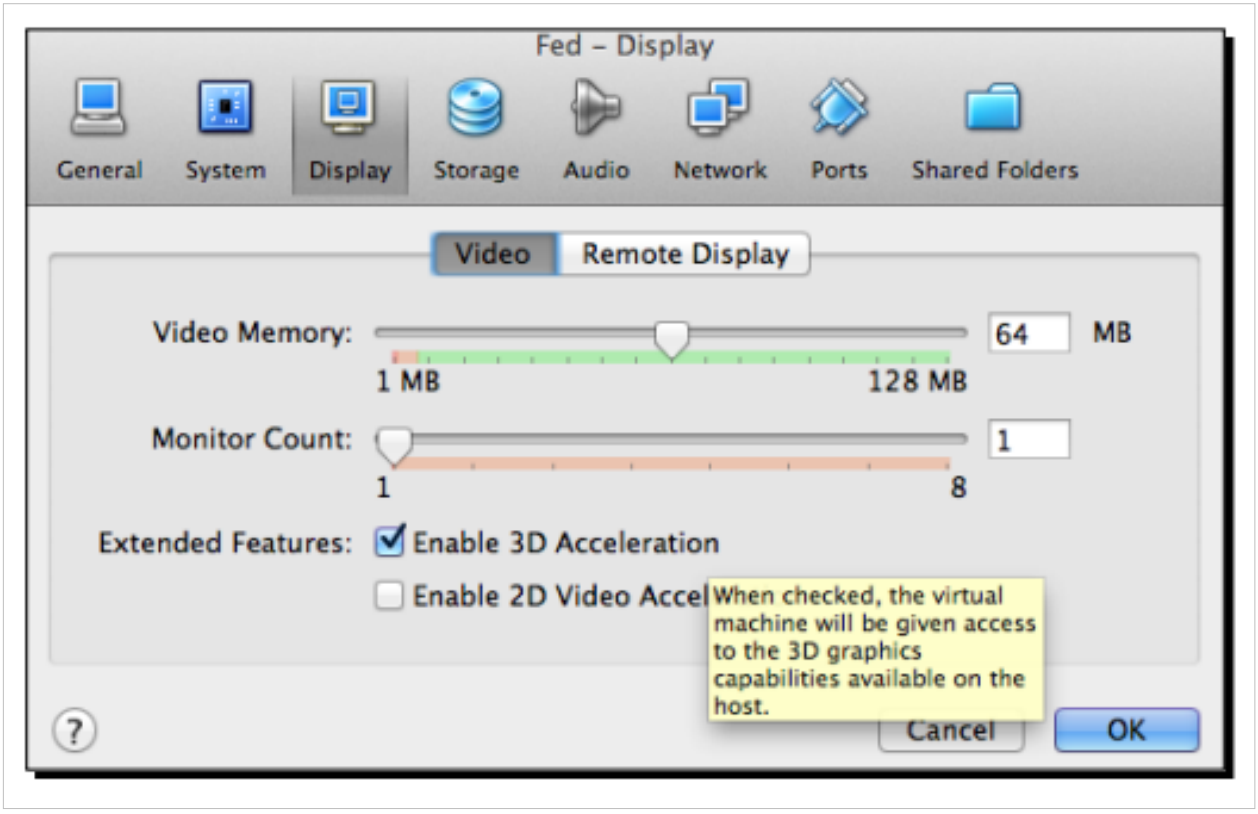
다행히 본 서적에서 GNOME 셸 UX를 심도 있게 다루는 부분은 소량에 불과하므로 본문을 따라오려면 GNOME 패널/폴백 UX로 작업하면 되겠다.

개발 요구사항

이러한 요구사항을 기반으로 소프트웨어를 개발하면 더 많은 컴퓨터 능력, RAM, 저장공간이 필요하게 될 것은 분명하다. 최소한 아래의 요구사항을 충족한다면 개발 과정이 순조롭게 진행될 것이다.

- Multicore 2 GHz CPU
- 램 4 GB
- 하드웨어 저장공간 500 GB, SSD 권장

좋은 소식이라고 하면 가상화를 이용해 GNOME3 애플리케이션을 쉽게 개발할 수 있다는 점이다. 가상 머신에 Linux 배포판을 설치함으로써 전환을 통해 그곳에서 발견되는 문제를 해결하면 배포판의 버전과 상관없이 애플리케이션을 GNOME3에서 순조롭게 빌드할 수 있을 것이다. GNOME 셸의 경우 자신이 선호하는 가상 머신 프로그램에서 3D 가속화 옵션을 꼭 활성화하도록 한다.

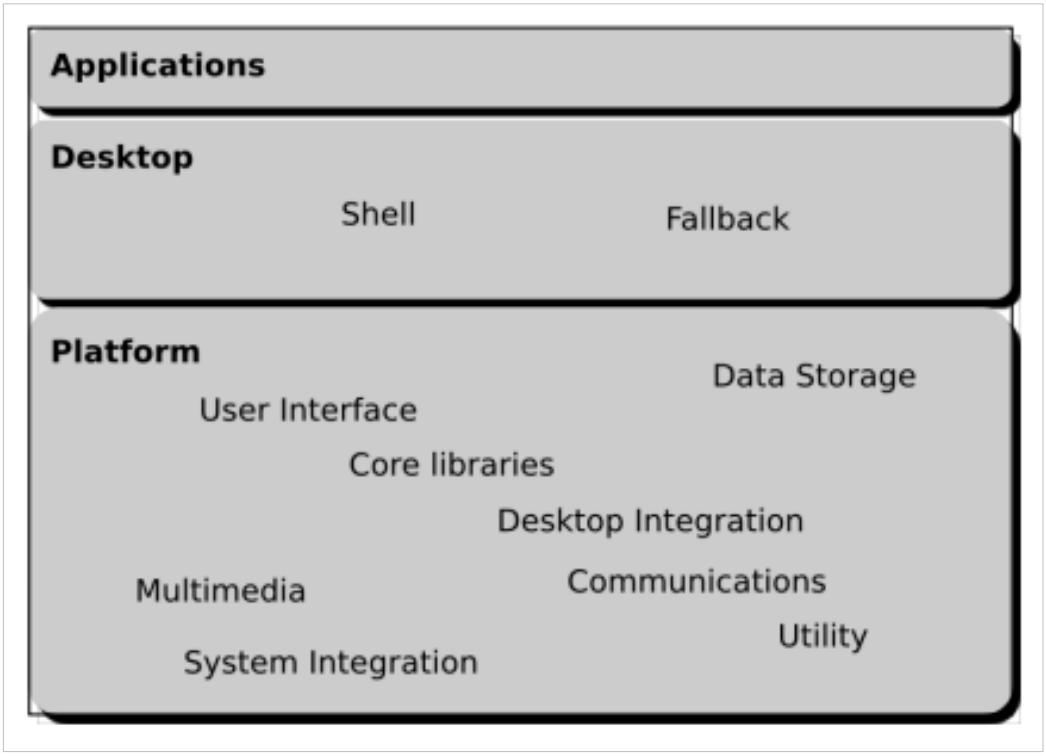


예를 들어 VirtualBox에서는 가상 머신의 Settings 대화상자로 들어가 Display 옵션을 클릭한 후 Enable 3D Acceleration 옵션을 체크한다. 단, 호스트 머신이 스스로 3D 가속화를 실행할 수 없는 경우 이 옵션을 활성화할 수 없음을 기억하라.

GNOME3 데스크톱 아키텍처

GNOME3를 논할 때 많은 사람들은 GNOME 셸만 지칭한다. 이것은 올바르지 않은 습관이다. GNOME 셸은 전체적인 GNOME 데스크톱 아키텍처의 일부에 불과하다. 따라서 (GNOME 패널/폴백 UX와 마찬가지로) 교체되기도, 심지어 제거되기도 한다.

사실 GNOME은 GNOME 셸 그 이상이다. GNOME은 애플리케이션의 기반 구조로서 시스템에게 말을 걸고, 텍스트를 멋지게 렌더링하고, 애니메이션을 유체로 표현하며, 데이터를 읽는 등의 일을 한다. 따라서 개발을 시작하기 전에 아키텍처를 먼저 잘 이해할 필요가 있다. 그래야만 설치해야 할 부분들을 이해하는 데에 도움이 될 것이다. 간소화한 GNOME 아키텍처 다이어그램을 살펴보자.



다이어그램을 살펴보면 애플리케이션과 함께 GNOME 셸이 GNOME 플랫폼 아키텍처 스택의 가장 위에 위치한다. 본 저서에서는 플랫폼 아키텍처를 다루면서 상위 계층에 대해서는 일부만 간단히 언급하도록 하겠다.

[	GNOME 플랫폼 참고자료는 GNOME 개발자 웹사이트, http://developer.gnome.org 에서 상세히 학습할 수 있다.	]

본 서적에서 다루게 될 GNOME 플랫폼의 구체적인 구성요소는 다음과 같다.

- 코어 라이브러리: 아키텍처의 최하위층 인터페이스에 해당한다. 아래의 항목들이 여기에 포함된다.
  - GObject: GNOME에서 객체 시스템에 해당한다. C 언어를 이용한 GNOME의 객체 지향 프로그래밍 접근법에 해당한다.
  - GLib: 다용도의 라이브러리로, 아키텍처 내 모든 부분들이 사용하는 기반 구조를 포함한다.
  - GIO: 가상 파일 시스템 라이브러리로, 파일, 볼륨(volumes), 드라이브에 대한 접근성을 추상적인 방식으로 제공한다.
- 사용자 인터페이스 라이브러리: 그래픽 애플리케이션을 빌드하기 위한 툴킷으로, 아래를 포함한다.
  - GTK+: 본래 GIMP 툴킷이라고 불렸으며, GNOME에서 그래픽 애플리케이션을 빌드하기 위한 기본 툴킷이다. 위젯과 툴 집합을 제공한다.

    - Cairo: 캔버스에 그림을 그리도록 도와주는 라이브러리다. 주로 새로운 위젯을 생성하거나 기존의 위젯을 확장하는 데에 사용된다.
    - Pango: 텍스트 렌더링을 실행하는 데에 도움이 되는 라이브러리다.
    - ATK: 접근성 툴킷이다. 서로 다른 요구를 가진 GNOME의 특수 사용자들에게 좋은 경험을 제공하는 방법을 제공한다.
    - Clutter: 다채로운 유체 애니메이션으로 애플리케이션을 생성하는 데에 사용되는 툴킷이다. OpenGL이 필요하다.
    - WebKit: 웹 툴킷이다. HTML5 문서를 표시할 수 있는, 모든 기능이 적용된 엔진을 제공한다.
- 멀티미디어 라이브러리: 멀티미디어 파일의 재생 및 저작(authoring)을 제공하며, 다음을 포함한다.
    - GStreamer: 강력한 멀티미디어 데이터다.
- 데이터 저장공간: 접근 데이터의 라이브러리를 제공한다.
    - 에볼루션 데이터 저장공간(Evolution Data Storage; EDS): libebook과 libecal을 포함하며, 자신의 Evolution에서 관리되는 캘린더와 주소록으로 접근성을 제공한다.

본문에 걸쳐 몇 가지 툴을 사용할 것인데, 그 목록은 아래와 같다.

- seed: GNOME 세계에서 JavaScript 인터프리터에 해당한다. GNOME 애플리케이션 스크립트를 개발 시 사용할 것이다. GNOME 셸을 설치할 때 자동으로 설치된 시드(seed)를 가질 것이므로 설치 시 명시적으로 표시되진 않을 것이다.
- vala: 최근에 만들어진 객체 지향 프로그래밍 언어로, 주로 GNOME 애플리케이션을 개발 시 사용된다.
- Anjuta: GNOME을 위한 통합 개발 환경 소프트웨어다.
- Glade: GTK+를 위한 그래픽 애플리케이션 레이아웃 디자이너다.
- Gtranslator: 애플리케이션 사용자 인터페이스를 로컬 언어로 해석하기 위한 툴이다.
- Devhelp: 참조 검색(reference lookup) 툴로, 애플리케이션을 개발 시 매우 유용하다.

이러한 부분들의 실제 패키지명은 표준화되지 않았기 때문에 앞에서 실제 패키지명이 위에서 소개한 리스트와 약간 다를 수 있음을 주목한다. 각 절에서 리스트를 구체적으로 논하도록 하겠다.

GNOME과 SDK

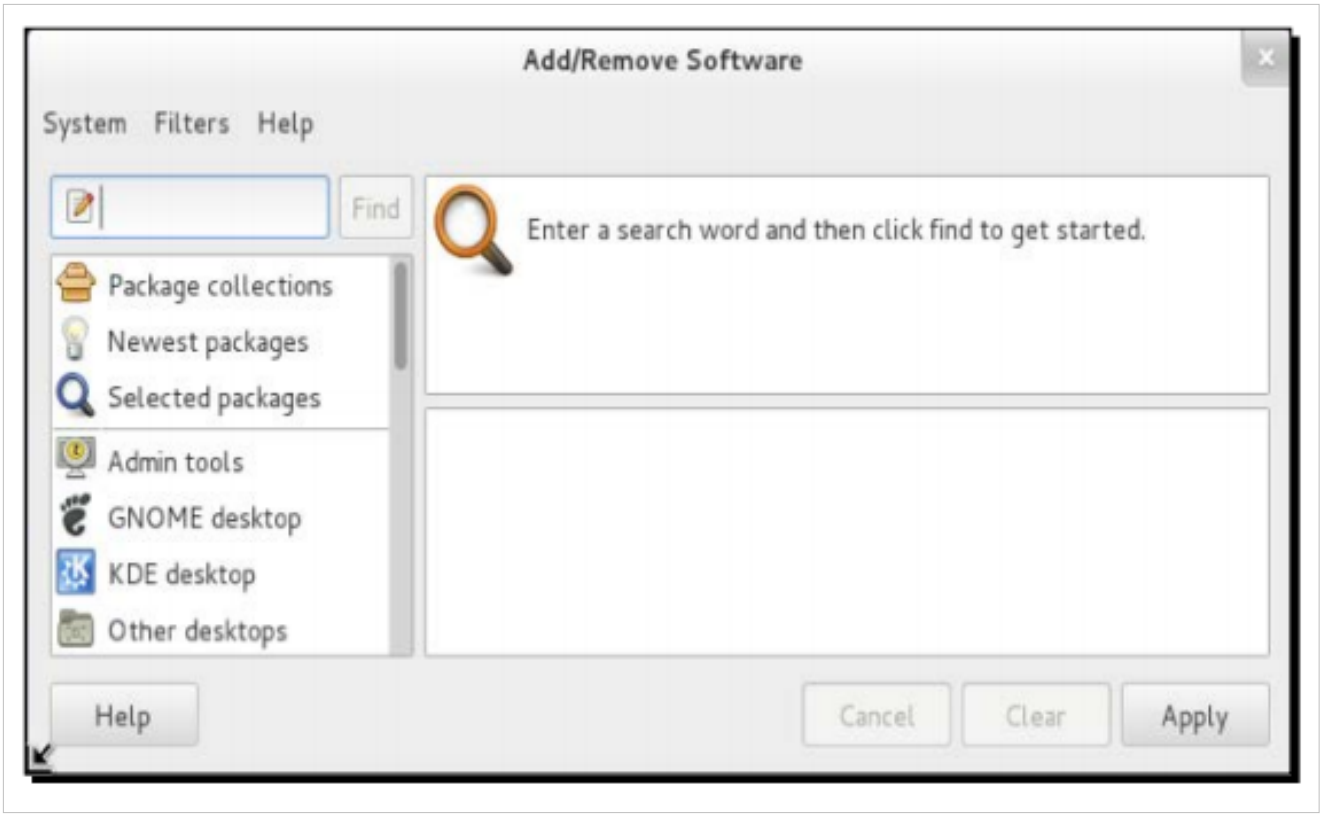
우리가 선호하는 Linux 배포판에 GNOME과 소프트웨어 개발 키트(SDK)를 설치할 것이다. 몇 가지 상용되는 배포판을 먼저 논하겠다. 하지만 이러한 배포판의 설치는 전혀 다루지 않을 것이므로 이미 설치를 완료하고 실행 중인 것으로 가정하겠다. GNOME과 SDK를 실행하기 위해서는 몇 가지 추가 패키지도 설치할 필요가 있다.

다른 설치 과정에 대한 설명은 무시하고 자신이 사용하는 배포판을 다룬 절로 건너뛰길 바란다. 자신이 사용하려는 배포판이 본문에 실려 있지 않더라도 걱정 말고 가장 가까운 변형(variant) 배포판을 선택하여 패키지명을 조정하면 되는데, 배포판을 설치할 때는 서치(search) 함수를 사용할 것이기 때문이다. 이제 소매를 걷고 본격적으로 시작해보자.

실행하기 - Fedora 17에서 GNOME과 SDK 설치하기

Fedora 17은 기본값으로 GNOME3를 사용하므로 여기서는 SDK의 설치에 중점을 두겠다. Fedora 17에 패키지를 설치하기 위해 Add/Remove Software 툴을 이용할 것이다. 그 과정은 다음과 같다.

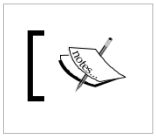
1. 화면의 상단 좌측 모서리에 위치한 **Activities** 버튼을 클릭한다.
2. **Applications** 버튼을 클릭한다.
3. **Add/Remove Software** 버튼을 클릭한다.
4. 아래와 같은 툴이 나타난다.



무슨 일이 일어났는가**?**

Add/Remove Software 툴은 우리가 살펴보고 선택할 수 있도록 Fedora 17 패키지의 데이터베이스를 표시한다. 화면의 좌측면에는 검색 텍스트박스가 표시되고, 아래에서는 컬렉션 패키지를 확인할 수 있다. 컬렉션 패키지는 유사한 기능을 수행하는 패키지의 그룹이다. 화면의 상단 좌측에는 컬렉션의 내용이나 검색 결과가 표시될 것이며, 이 영역에서 선택한 패키지의 설명은 그 아래 박스에 표시된다.

SDK는 소위 개발 패키지라고 불리는 형태로 되어 있다. Fedora에서 개발 패키지의 이름에는 -devel 이라는 단어가 뒤에 붙는다. 실제 라이브러리는 뒤에 -devel가 붙지 않은 패키지에 위치하며, devel 패키지가 설치될 때 기본적으로 설치된다. 가령 GLib는 glib2 패키지에서 이용 가능하며, 그것이 지원하는 개발 패키지는 glib2-devel 패키지에서 이용 가능하다. glib2-devel 패키지가 설치될 때마다 glib2 패키지가 자동으로 설치된다.

[	개발 패키지라 함은 헤더 헤더 파일을 비롯해 개발 시에만 필요하며 실행 시에는 필요하지 않은 파일들이 포함되어 있음을 의미한다.	]

이제 개발 패키지를 함께 찾아보도록 하자.

설치할 패키지 표시하기

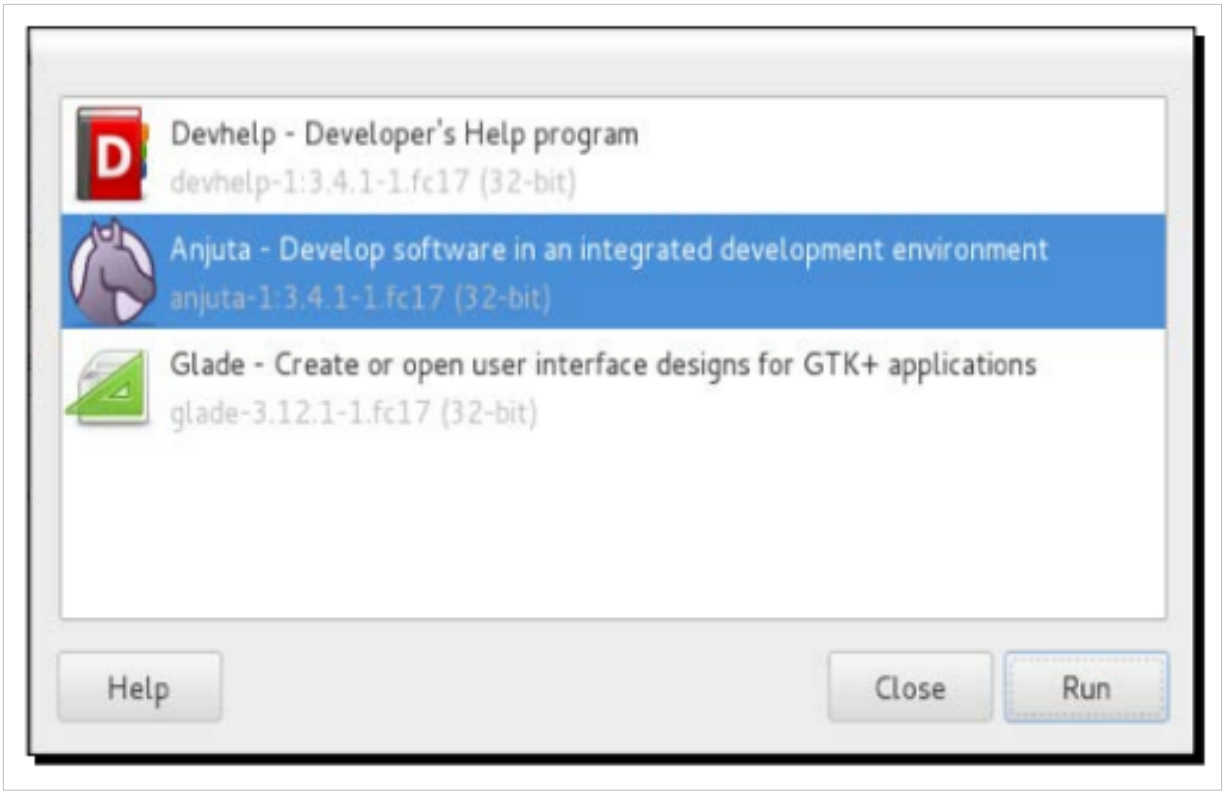
이 툴을 이용해 패키지를 설치하기 전에 설치할 패키지를 표시해야 한다. 검색창(search box) 제일 앞에 glib2-devel을 입력하라. 그리고 Find 버튼을 클릭하면 결과가 표시될 것이다. 여러 개의 행 중에 glib2-devel 문자열 다음에 긴 버전 번호(필자의 경우 glib2-devel-2.32.1-1.fc17)가 따라오는 행이 발견될 것이다. 체크박스를 왼쪽으로 넘긴다. 그러면 툴은 아이콘에 큰 덧셈 기호를 표시할 것인데, 이는 설치 표시가 되었다는 의미다. 체크박스가 보이지 않으면 이미 설치되었단 뜻이다.

아래 표는 Fedora 17에서 패키지명과 앞의 아키텍처 절에서 설명한 GNOME 구성요소 간 매핑을 제공한다. 문서 패키지도 설치하고자 한다 (이름 뒤에는 -doc 또는 -docs가 무작위로 붙는다). 책 전체에 걸쳐 사용할 툴과 기본 개발 패키지도 설치할 예정이다. 따라서 검색창에서 이러한 패키지들을 검색하여 체크박스를 표시하도록 하자.

서브시스템	패키지명
코어 라이브러리	glib2-devel (GIO와 GObject는 이미 해당 패키지 내에 있다)
사용자 인터페이스 라이브러리	gtk3-devel gtk3-devel-docs cairo-devel pango-devel atk-devel clutter-devel clutter-doc webkitgtk3-devel webkitgtk3-doc
멀티미디어 라이브러리	gstreamer-devel gstreamer-devel-docs
데이터 저장공간	evolution-data-server-devel evolution-data-server-doc
툴과 기본적인 개발 패키지	vala vala-doc vala-tools anjuta glade (don't mix up with glade3!) glade-libs gtranslator devhelp

패키지 설치 준비하기

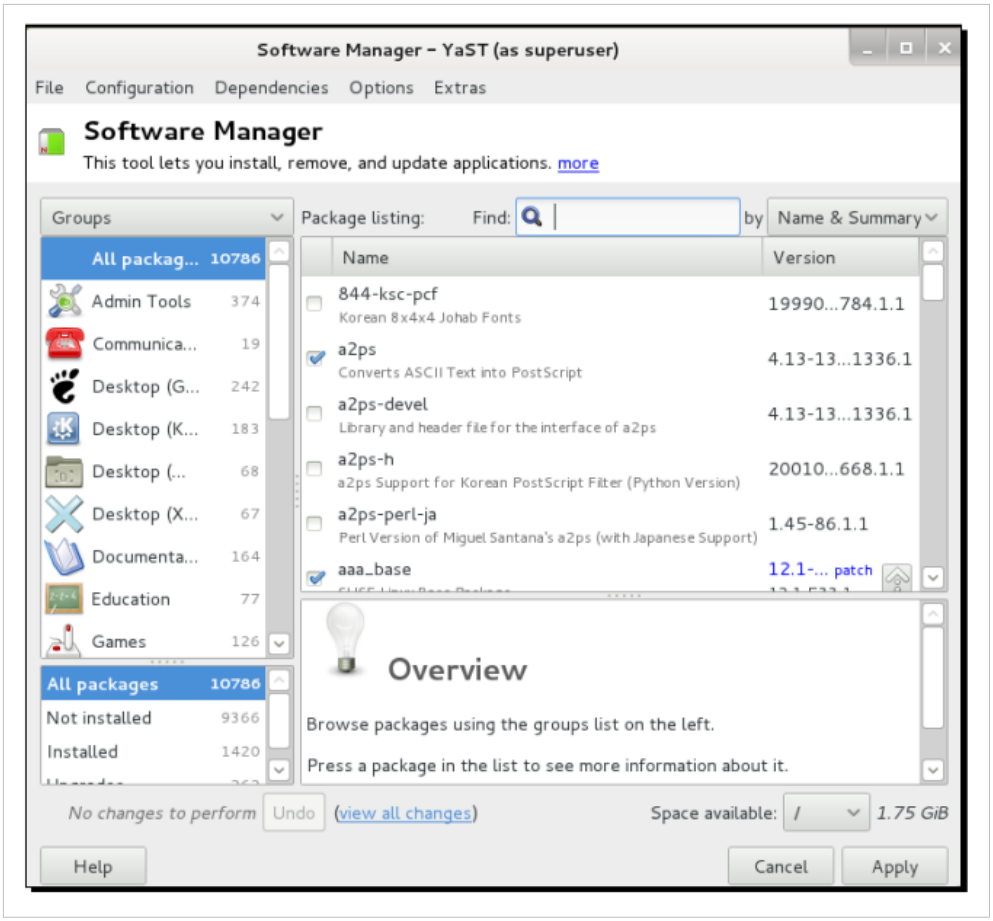
패키지를 표시했다면 이제 설치할 준비가 되었다. 화면 하단에 Apply 버튼을 클릭하기만 하면 된다. 앞서 선택한 패키지가 필요로 하는 추가 패키지의 설치도 원하는지 묻는 대화상자가 나타날 것이다. 동의하려면 Continue 버튼을 클릭하여 노드 제스처(node gesture)를 만든다. 그 다음 패키지를 다운로드하여 시스템으로 설치하는 동안 루트 비밀번호를 입력하고 잠시 기다린다. 설치가 모두 끝나면 아래 스크린샷과 같이 새로운 창이 뜨면서 실행이 가능한 새로 설치된 애플리케이션이 표시되는데, 이 대화상자는 다음 장에서 살펴볼 것이다.



실행하기 - openSUSE 12에서 GNOME과 SDK 설치하기

openSUSE 12에는 GNOME이 기본적으로 포함되어 있으므로 전혀 염려할 필요 없이 SDK 설치에 집중하면 된다. 애플리케이션을 관리하기 위해 openSUSE는 Yet another Setup Tool(YaST) 툴을 제공한다. openSUSE 12에서 SDK를 설치하기 위해서는 다음 단계를 따른다.

1. 화면의 상단 좌측 모서리에 위치한 **Activities** 버튼을 클릭한다.
2. **Applications** 버튼을 클릭한다.
3. **YaST** 버튼을 클릭한다 (또는 텍스트 필드에 **YaST** 를 타이핑한다).
4. **YaST** 가 열렸다; 계속하려면 **Software Management** 아이콘을 클릭한다.
5. 서버로 접속하고 패키지 데이터베이스를 새로고침하고 나면 사용할 준비가 되었다. 그 모습은 다음과 같다.



무슨 일이 일어났는가**?**

YaST는 openSUSE에서 시스템 관리 툴의 집합체로, Software Manager는 그러한 툴들 중 하나에 해당한다. 그 명칭에서 알 수 있듯이 이 툴을 이용해 GNOME SDK를 설치할 것이다. 그림에서 볼 수 있듯이 툴에는 두 개의 메인 열이 있는데, 각 열은 두 개의 섹션으로 나뉜다. 왼쪽 열은 패키지 범주를, 오른쪽 열은 선택된 범주의 내용(또는 검색 결과)과 패키지에 대한 설명을 표시한다. 검색 기능만 이용해 툴을 이용하고 왼쪽의 범주는 건들지 않을 예정이다.

Fedora에서 **Add/Remove Software** 와 유사하게 **YaST** 도 표시 후 설치(mark-and-install)하는 방식을 이용하므로 소프트웨어를 설치하기 전에 표시할 필요가 있다. 이 개념을 이용해 우리는 원하는 소프트웨어를 선택하고 나서 버튼을 클릭하기만 하면 선택한 소프트웨어가 설치된다. 한 번 해보자.

SDK 패키지 표시하기

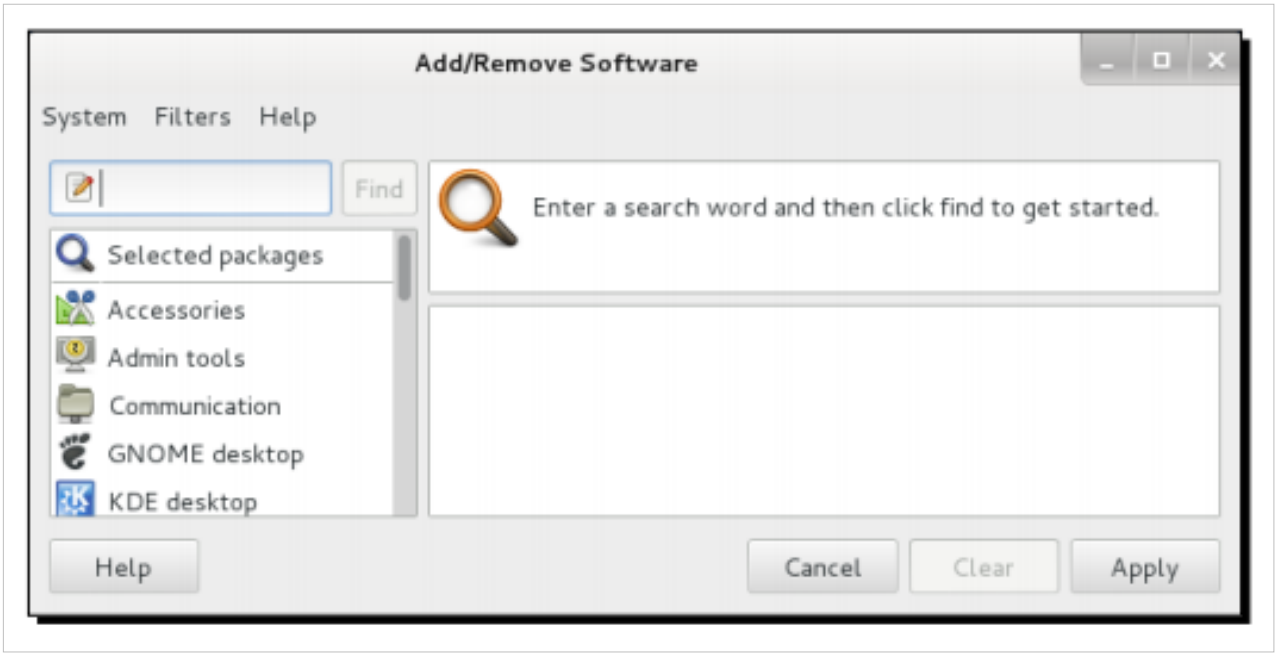
SDK는 여러 다른 개발 패키지에 흩어져 있다. RPM 패키지 관리 시스템을 사용하는 Fedora와 비슷하게 개발 패키지명 뒤에는 -devel이 붙으며, 그에 상응하는 라이브러리는 개발 패키지가 설치되면서 자동으로 설치될 것이다.

아래 소개된 표는 openSUSE 패키지명과 앞의 아키텍처와 관련된 절에 설명한 GNOME 구성요소 간 매핑을 제공한다. 검색창에 이 이름을 입력하여(한 번에 하나씩) 표시되는 검색 결과 엔트리의 좌측에 위치한 체크박스를 표시하면 된다. 체크박스를 표시한다는 것은 패키지가 설치 큐(installing queue)로 추가됨을 의미한다. 체크박스가 이미 표시되어 있다면 패키지가 이미 설치되어 있다는 뜻이다.

서브시스템	패키지명
코어 라이브러리	glib2-devel (GIO and GObject are already inside this package)
사용자 인터페이스 라이브러리	gtk3-devel gtk3-devel-docs libseed-gtk3-devel cairo-devel pango-devel atk-devel clutter-devel libwebkitgtk3-devel
멀티미디어 라이브러리	gstreamer-0_10-devel gstreamer-0_10-doc gstreamer-0_10-fluendo-mp3
데이터 저장공간	evolution-data-server-devel evolution-data-server-doc
툴과 기본적인 개발 패키지	vala anjuta glade (don't mix up with glade3!) gtranslator devhelp

설치 시작하기

패키지를 표시하고 나면 Apply 버튼을 눌러 설치를 시작하자. 모든 패키지가 설치될 때까지 기다렸다면 다음 장으로 넘어갈 준비가 끝났다.



실행하기 - Debian Testing에서 GNOME과 SDK 설치하기

Debian Testing(Wheezy라고도 불림)은 GNOME3를 기본 데스크톱으로 사용하므로 SDK의 설치만 중점으로 살펴보겠다. Fedora와 마찬가지로 Debian Testing도 패키지 관리의 실행에 Add/Remove Software 툴을 이용한다. 툴의 실행으로 시작해보자.

1. 화면의 상단 좌측 모서리에 위치한 **Activities** 버튼을 클릭한다.
2. **Applications** 버튼을 클릭한다.

3. **Add/Remove Software** 버튼을 클릭한다.
4. 아래와 같은 툴이 표시될 것이다.

무슨 일이 일어났는가**?**

Add/Remove Software 툴은 표시 후 설치하는 패키지 관리 툴이다. 즉, 실제 설치 과정을 실행하기 전에 설치하고자 하는 패키지를 선택할 기회가 주어진다는 의미다. 설치는 **Apply** 버튼을 눌러 확인을 할 때까지 지연된다. 이러한 설치 스타일을 이용하면 설치 과정에 방해 받지 않고 패키지를 살펴볼 수 있다.

위의 스크린샷에서 볼 수 있듯이 툴에는 두 개의 메인 열이 있다. 왼쪽 열은 검색창과 패키지 범주를 표시한다. 오른쪽 열은 검색 결과(또는 범주 내용)를 표시하고, 그 아래에는 패키지 설명을 보여주는 박스가 위치한다.

SDK 패키지 표시하기

SDK 패키지는 많은 개발 패키지에 따라온다. 개발 패키지는 헤더를 비롯해 컴파일과 연결(linking) 시 필요한 지원 파일을 모두 포함한다. 라이브러리로 연결된 애플리케이션을 실행하기만 할 경우 굳이 해당 패키지들을 모두 설치할 필요는 없다.

이러한 패키지들은 앞에 lib와 뒤에 -dev라는 단어가 붙어 명명된다. 그리고 애플리케이션이 필요로 하는 실제 라이브러리에 대해 내부 종속성(internal dependency)을 갖기 때문에 우리가 개발 패키지를 설치할 때마다 실제 라이브러리를 포함하는 패키지도 자동으로 따라간다.

아래는 아키텍처에 관한 절에서 설명한 GNOME 구성요소와 Debian 패키지 간 매핑을 나타낸다. 패키지명을 검색창에 입력하고 설치할 패키지를 표시해보자.

서브시스템	패키지명
코어 라이브러리	libglib2.0-dev 9 (GIO and GObject are already inside this package) libglib2.0-doc
사용자 인터페이스 라이브러리	libgtk-3-dev libgtk-3-doc libcairo2-dev libcairo2-doc libpango1.0-dev libpango1.0-doc libatk1.0-dev libatk1.0-doc libclutter-1.0-dev libclutter-1.0-doc libwebkitgtk-3.0-dev libwebkitgtk-3.0-doc
멀티미디어 라이브러리	libgstreamer0.10-dev
툴과 기본적인 개발 패키지	valac-0.16 anjuta glade gtranslator devhelp

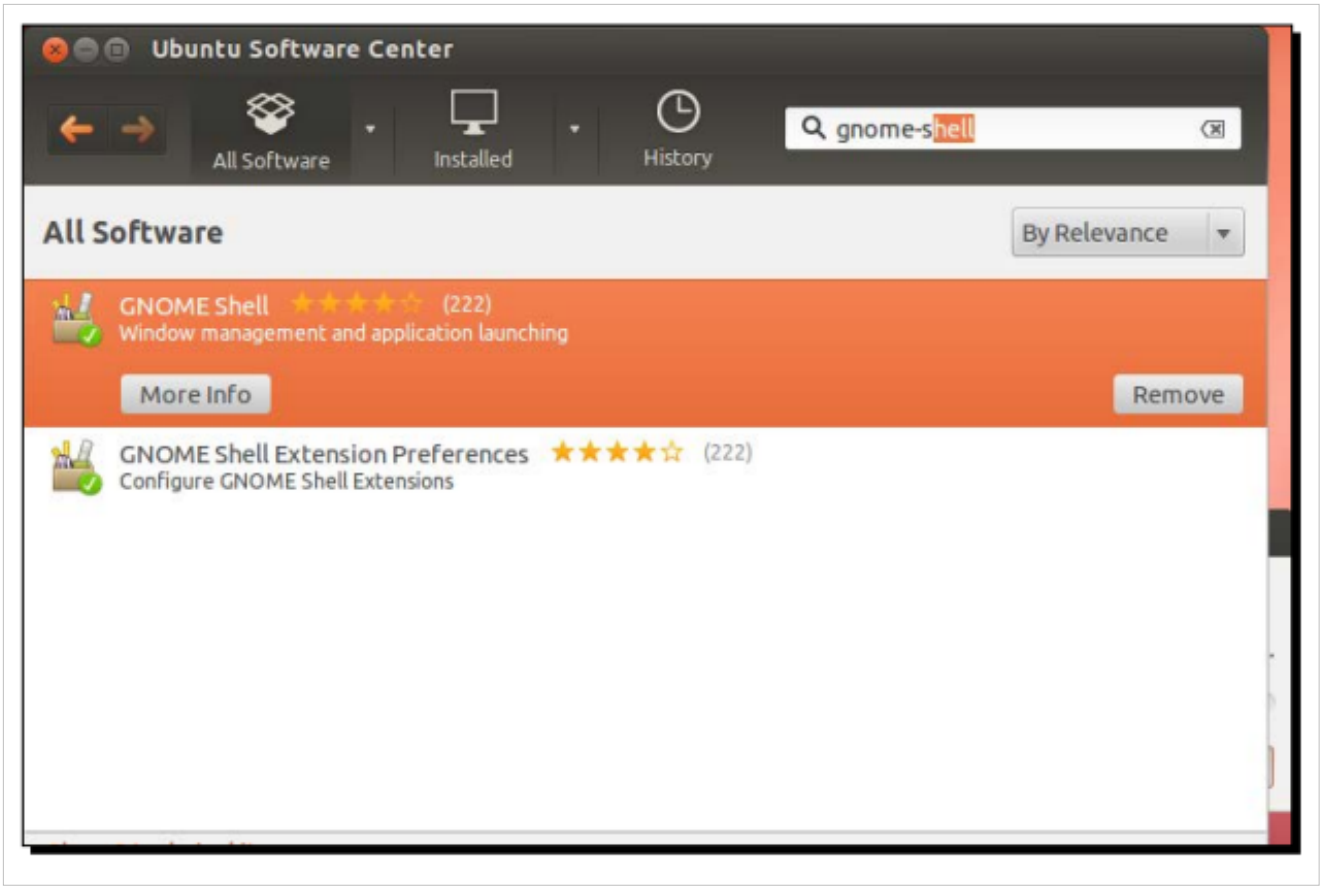
설치 적용하기

패키지를 표시한 후에는 **Apply** 버튼을 눌러 설치를 시작한다. 잠시 쉬면서 구경하자. 모든 패키지가 설치될 때까지 기다린다. 설치가 끝나면 다음 장으로 넘어갈 준비가 된 것이다.

실행하기 - Ubuntu 12.0에서 GNOME과 SDK 설치하기

Ubuntu에는 고유의 데스크톱, **Unity**가 따라온다. **Ubuntu Software Center** 툴을 이용해 GNOME 3 데스크톱과 SDK를 설치하는 것이 목적이므로, 아래 과정을 실행한다.

1. 화면의 상단 좌측 모서리에 위치한 **Ubuntu Software Center** 아이콘을 클릭한다.
2. 검색창 안을 클릭하고 **gnome-shell** 을 입력한다.
3. 첫 번째 검색 결과에서 **Install** 을 클릭한다.
4. GNOME Shell 이 설치되었다.



무슨 일이 일어났는가?

GNOME 셸 데스크탑을 설치하였다. GNOME 패널 또한 내부적으로 명시된 패키지 종속성 형태로 GNOME 셸과 연결되기 때문에 두 가지 UX 모두 설치할 것이다.

Ubuntu Software Center 툴은 표시 후 설치하는 방식을 이용하지 않으므로 검색 결과마다 설치를 따로 실행해야 한다. 설치 과정은 배경에서 이루어지지만 아쉽게도 설치 도중에 검색을 할 수가 없다. 하지만 설치하길 원하는 다른 패키지가 동일한 결과 페이지에 나타나면 설치 과정에 대기하도록(queue) **Install** 버튼을 클릭할 수 있다.

SDK 설치 계속하기

Ubuntu는 Debian에서 파생되었으므로 패키지 명명 시스템이 동일하다. Ubuntu 패키지와 Debian 절에서 설명한 GNOME 구성요소 간 매핑을 살펴보자. 맵에 설명된 모든 패키지를 하나씩 설치해보자.

패키지가 모두 설치되면 다음 장으로 넘어갈 준비가 되었다!

요약

이번 장에서는 시스템 요구사항, 기본 아키텍처, GNOME과 SDK의 설치에 관해 많은 내용을 학습하였다. 그 중에서 GNOME3는 두 개의 UX를 제공함을 학습하였다. 이를 실행하고 개발하기 위한 시스템 요구사항을 이해했다. 뿐만 아니라 가상 환경에서 작업할 준비를 시키는 데에 도움이 되는 팁을 배웠다. 기본 GNOME 데스크톱 아키텍처와 그 내부에 있는 각 구성요소에 대한 설명을 비롯해 본문에 걸쳐 사용할 툴도 모두 설명하였다.

특정 GNOME 구성요소가 아키텍처 다이어그램에서 어디에 위치하는지도 이해했을 것이다. 또 어떤 툴을 사용할 예정인지, 유명한 Linux 배포판에서는 어떻게 포함되는지도 이해할 것이다. 뿐만 아니라 이러한 툴의 설치 방법도 학습하였다. 이러한 지식은 Linux 배포판이 시스템을 어떻게 이용하는지에 대한 통찰력을 제공한다.

GNOME3에서 애플리케이션을 개발하는 방법을 학습할 준비가 충분히 되었다. 이제 개발 툴을 준비할 단계다. 다음 장에서는 개발 툴을 준비시키고, 프로그래밍 언어들 중에서 본문에 걸쳐 사용할 Vala를 간단히 언급하겠다.

Notes

GNOME3ApplicationDevelopmentBeginnersGuide:Chap 02

제 2 장 무기 준비하기

무기 준비하기

프로젝트에 사용되는 툴에 대한 친숙도는 성공적인 소프트웨어 프로젝트에 기여하는 주요 요인들 중 하나다. 본문에서 사용하는 툴은 우리의 무기로 간주해도 좋다. 올바로 무기를 사용하면 대상(우리가 생성하는 소프트웨어)을 올바로 무너뜨릴 뿐만 아니라 효율적으로 사용함으로써 시간을 절약할 수도 있다. 그렇지 않으면 역효과를 발생하여 오히려 해가 되기도 한다.

이번 장에서는 본문에 걸쳐 사용하게 될 주요 툴을 익히고자 한다. 그 툴로는 Anjuta라고 불리는 통합 개발 환경**(IDE)**, Glade라는 인터페이스 레이아웃 디자이너, Devhelp라는 개발 참조 브라우저가 해당한다. 이러한 툴들은 초보자들에게 엄청난 도움이 되며, 서적을 모두 끝내고 전문가가 되더라도 여전히 유익하게 활용될 것이다.

이번 장에서는 아래와 같은 주제들을 살펴볼 것이다.

- Anjuta에서 프로젝트 생성하기
- IDE 레이아웃
- 소스 코드 탐색 및 조작
- 프로젝트 빌드 관리하기
- Glade를 이용한 애플리케이션 레이아웃의 디자인을 위해 실물모형(mockup) 이용하기
- Devhelp를 이용해 참고 서적 읽기

이제 시작해보자!

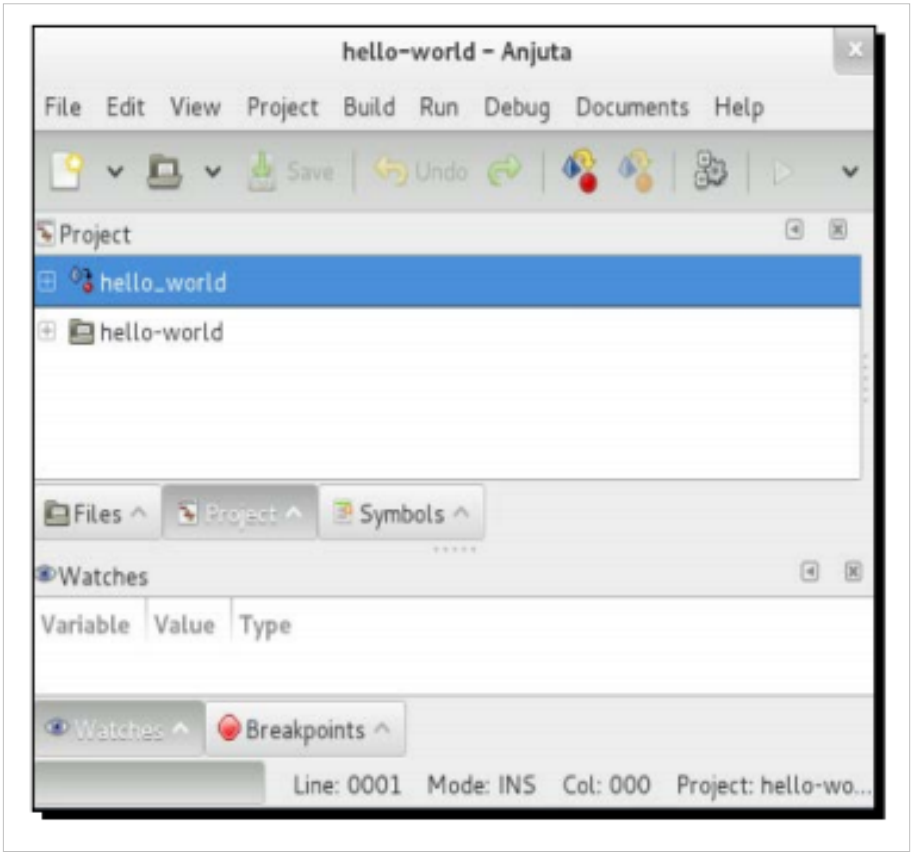
Anjuta 작동시키기

이번 장에서는 Anjuta를 사용해보도록 하겠다. GNOME Shell에서는 Activities | Anjuta를, GNOME Panel에서는 Applications | Programming | Anjuta 를 탐색하여 Anjuta를 호출해보자.

실행하기 - 새로운 Vala 프로젝트 생성하기

가벼운 연습 삼아 간단한 Vala 프로젝트를 생성해보도록 하겠다. 이 프로젝트를 이용해 Anjuta의 기능을 살펴볼 예정이다. 다음을 실행한다.

1. Anjuta의 메인 화면의 **Create a new project** 를 클릭한다.
2. 대화상자가 나타나면 **Vala** 탭을 클릭한다.
3. **GTK+(Simple)** 옵션을 클릭한 후 **Continue** 를 클릭한다.
4. 프로젝트 정보 대화상자를 채우고 **Project Name** 필드에 hello-world 를 입력한 후 나머지 내용을 채우고 나서 **Continue** 를 클릭한다.
5. 다음으로 제시된 몇 가지 질문은 답하지 않고 두어라(프로젝트 경로 등의 내용 변경을 원하는 경우는 제외). **Use GtkBuilder for user interface** 체크박스가 표시되었는지 확인한다. **Continue** 를 클릭한다.
6. 확인 대화상자에 표시된 정보가 마음에 들면 **Apply** 를 누른다.
7. 새로운 프로젝트가 아래 화면과 같이 생성되었다.



무슨 일이 일어났는가**?**

Anjuta에서는 소프트웨어를 구성하는 모든 파일과 자원의 컨테이너에 해당하는 프로젝트를 생성할 것이다. 프로젝트 빌드와 배치(deployment)에 필요한 소스 코드, 파일, 이미지 등을 추가할 것이다. 방금 우리는 Vala 프로그래밍 언어를 이용해 간단한 GTK+ 프로젝트를 생성하였다. 지금 당장 Vala가 무엇인지 모른다거나 본문에 제시된 코드를 이해하지 못한다 하더라도 걱정하지 않아도 되는데, 이번 장을 완료하고 학습할 것이기 때문이다.

방금 실행한 마법사는 생성된 소스 코드와 사용자 인터페이스를 포함해 파일 집합을 생성하였다. 이러한 파일들은 템플릿으로 간주되기도 하는데, 개발을 진행 시 이러한 파일들을 변경할 것이다.

Anjuta는 IDE로서 거의 모든 기능을 장착하고 있다고 볼 수 있다. 소스 코드와 일반 파일의 에디터, 디버거, Glade, 사용자 인터페이스 레이아웃 디자이너를 갖고 있다. 다른 IDE에 익숙하다면 뷰에 약간의 차이가 눈에 띌 것인데, 어서 분석해보도록 하자.

IDE 레이아웃

IDE의 레이아웃은 꽤 간단하며, 툴바, 에디터, 배열이 가능한 Dock(dock)로 구성된다. 툴바 또한 간단하고 익숙한 요소다. 툴바는 파일 및 편집 연산, 프로젝트 실행과 빌드, 디버깅 연산과 같은 특정 함수의 단축키에 해당한다.

Anjuta에는 두 가지 종류의 에디터가 있다. 하나는 소스 코드 에디터, 그리고 나머지는 사용자 인터페이스 에디터인 Glade가 그것이다. 소스 코드 에디터를 먼저 알아보자.

여느 고급 소스 코드 에디터와 마찬가지로 Anjuta는 꽤 강력한 에디터를 제공한다. 눈에 띄는 기능으로는 다음이 포함된다.

- 행 번호 매기기**:** 에디터의 왼쪽에 행 번호가 표시된다. 현재 행의 행 번호는 볼드체로 표시되어 지금 어디에 있는지 쉽게 알 수 있다.
- 구문 강조**:** 소스 코드 내 토큰(token)은 색이 칠해져 변수, 클래스, 주석 등을 쉽게 구별할 수 있다.
- 코드 완성**:** Anjuta는 양호한 코드 완성 기능을 제공한다. 객체 인스턴스를 입력하면 가능한 클래스 member 또는 메서드의 리스트가 표시된다.

Docks 는 특정 함수를 포함하는 툴의 집합이다. View 메뉴에서 개별적으로 토글하여 표시/숨김을 결정할 수 있다. 하나씩 간단히 소개해보자.

북마크

해당 툴은 우리가 편집 중인 파일 내에서 위치를 북마크하도록 해준다. 소스 코드가 커지고 복잡해질수록 이 툴이 유용해진다. 하나 이상의 파일을 편집하거나 상호 참조(cross referenced)될 때마다 북마크 엔트리를 클릭하면 파일 간 빠른 전환이 가능하다.

파일

이 툴은 프로젝트 내 모든 파일을 열거한다. 프로젝트 폴더 내 파일들을 빠르게 살펴보는 파일이다. 파일의 편집을 원할 경우 관심이 있는 파일을 더블 클릭하면 에디터가 나타날 것이다.

프로젝트

이 툴은 프로젝트에 대한 우리의 뷰 역할을 한다. 이는 범주적으로 프로젝트의 내용을 열거한다. 어떤 파일이 사용자 인터페이스 파일이고, 어떤 파일이 소스 코드인지 쉽게 확인이 가능하다. 프로젝트 항목을 더블 클릭하면 에디터(사용자 인터페이스 레이아웃 파일이나 소스 코드) 또는 대화상자(그 외 프로젝트 항목)를 불러올 것이다. 여기서 우리는 Anjuta를 비롯해 프로젝트 항목의 많은 측면들, 즉 설치 경로, 컴파일러 스위치 등을 제어할 수 있다.

기호

프로젝트 내 모든 기호를 표시한다. **File**, **Project**, **Search** 탭은 관심이 있는 기호의 범위를 나타낸다. 기호를 더블 클릭하면 에디터는 기호가 위치한 행으로 즉시 이동한다.

Watches

관심이 있는 모든 변수의 값을 표시하는 디버거 툴이다. "debugging-by-doing-printf" 를 즐기는 사람이라면 이 툴이 낫다고 생각할 것이다. 불행하게도 이 툴은 C 언어로 개발할 때만 작동한다. 하지만 Vala에서 생성한 C 코드와 함께 이용해도 좋다.

Breakpoint

Breakpoint를 관리하는 디버거 툴이다. Breakpoint는 소스 코드 내에서 프로그램을 실행할 때 잠시 정지하길 (pause) 원하는 장소를 나타낸다. Breakpoint와 Watches는 삶을 훨씬 수월하게 해주는 유용하고 강력한 툴이다. Watches Dock와 마찬가지로 C 언어로 개발 시에만 작동한다.

메시지

해당 Dock는 컴파일러, 링커, 그 외 빌드 툴의 메시지를 표시한다. 프로젝트의 빌드를 시작 시 활성화된다.

Terminal

Terminal 셸을 포함하는 Dock다. 셸에 직접적으로 명령을 호출하는 것이 가능하다. 프로젝트를 성공적으로 빌드하면 활성화된다.

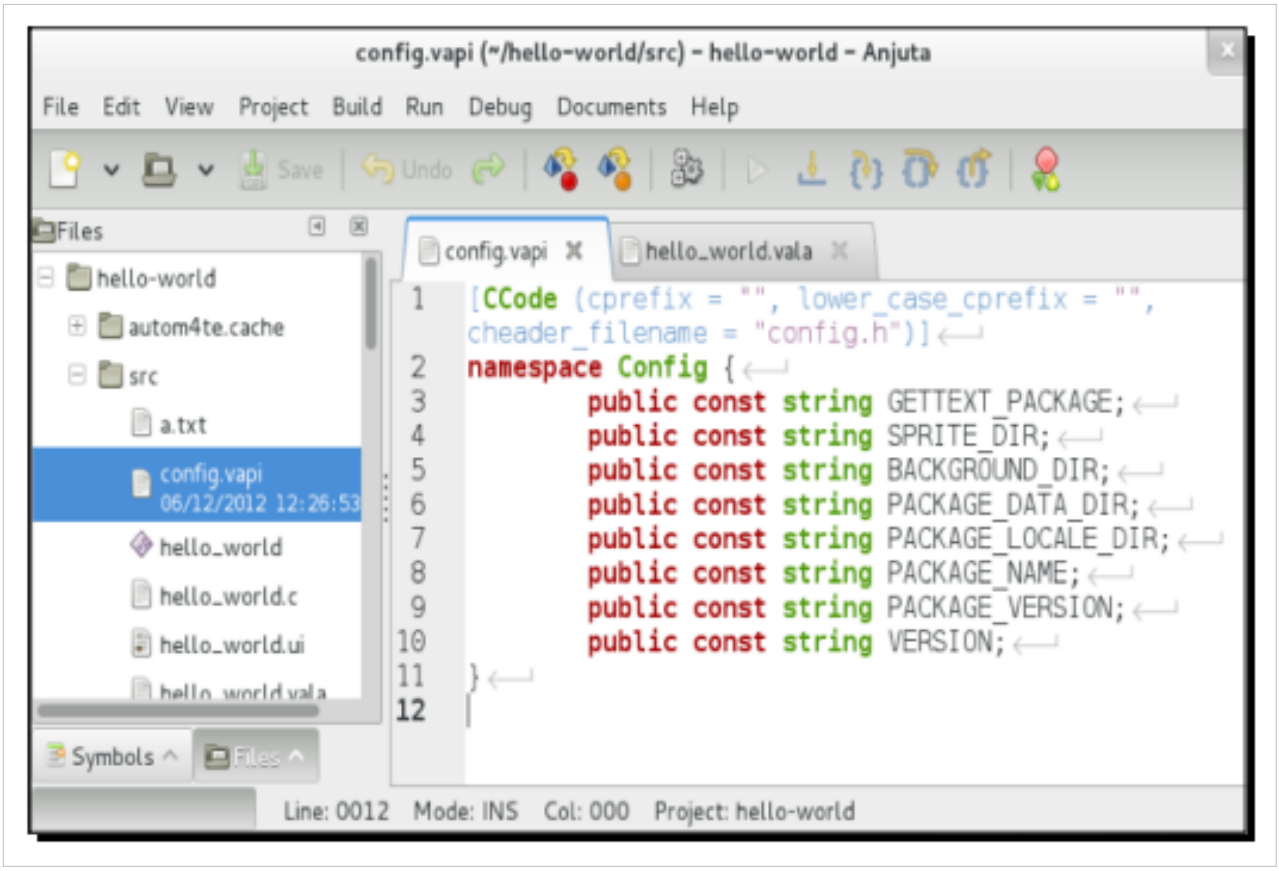
탭들 간 탐색

에디터는 동시에 다수의 파일을 편집할 수 있다. 우리가 편집 중인 파일은 탭에 표시되며 쉽게 전환이 가능하다.

실행하기 - 탭들 간 탐색하기

프로젝트로부터 여러 개의 파일을 열고 파일 간 탐색하는 방법을 살펴보자. 복잡한 프로젝트를 편집하는 중인데 파일이 서로 의존하고 있다는 시나리오일 경우, 하나의 파일을 편집하면 다른 파일도 변경해야 할 것이다.

1. **View** 메뉴를 클릭하고 **Files** Dock 가 활성화되어 있도록 하라.
2. **Files** Dock를 클릭하고 **src/** 를 찾을 때까지 리스트를 탐색하라.
3. 플러스 버튼을 클릭하여 리스트를 확장시켜라.
4. **hello_world.vala** 를 더블 클릭하라.
5. **config.vapi** 를 더블 클릭하라.
6. 이제 두 개의 파일이 열렸을 것이며, 키보드에서 *Alt+1* 과 *Alt+2* 키 조합을 누르면 탭들 간 전환이 가능하다. 더 많은 파일을 열려면 *Alt+3*, *Alt+4* 등을 눌러 탐색한다.



무슨 일이 일어났는가**?**

우리가 파일을 열 때마다 Anjuta는 파일을 탭 안에 넣는다. 탭은 다수의 파일을 수반하는 개발 시 꽤 유용하다. 보통 우리는 동시에 하나 이상의 파일을 열게 될 것이다. 여기서 논한 기능은 이러한 파일 간 전환을 수월하게 해준다.

주석 블록

Anjuta는 소스 코드 내 텍스트 블록을 주석 블록으로 포함시키는 간편한 방법을 제공한다. 그 반대도 가능하여 텍스트 블록으로부터 주석 마커를 제거하는 수도 있다. 주석 처리(comment out)의 효과를 확인하고 싶거나 의심이 가는 코드 블록이 있다고 가정해보자. 이러한 기능은 코드 블록의 주석 처리와 주석 제거를 도와주며, 특히 블록의 규모가 클 때 빛을 발한다.

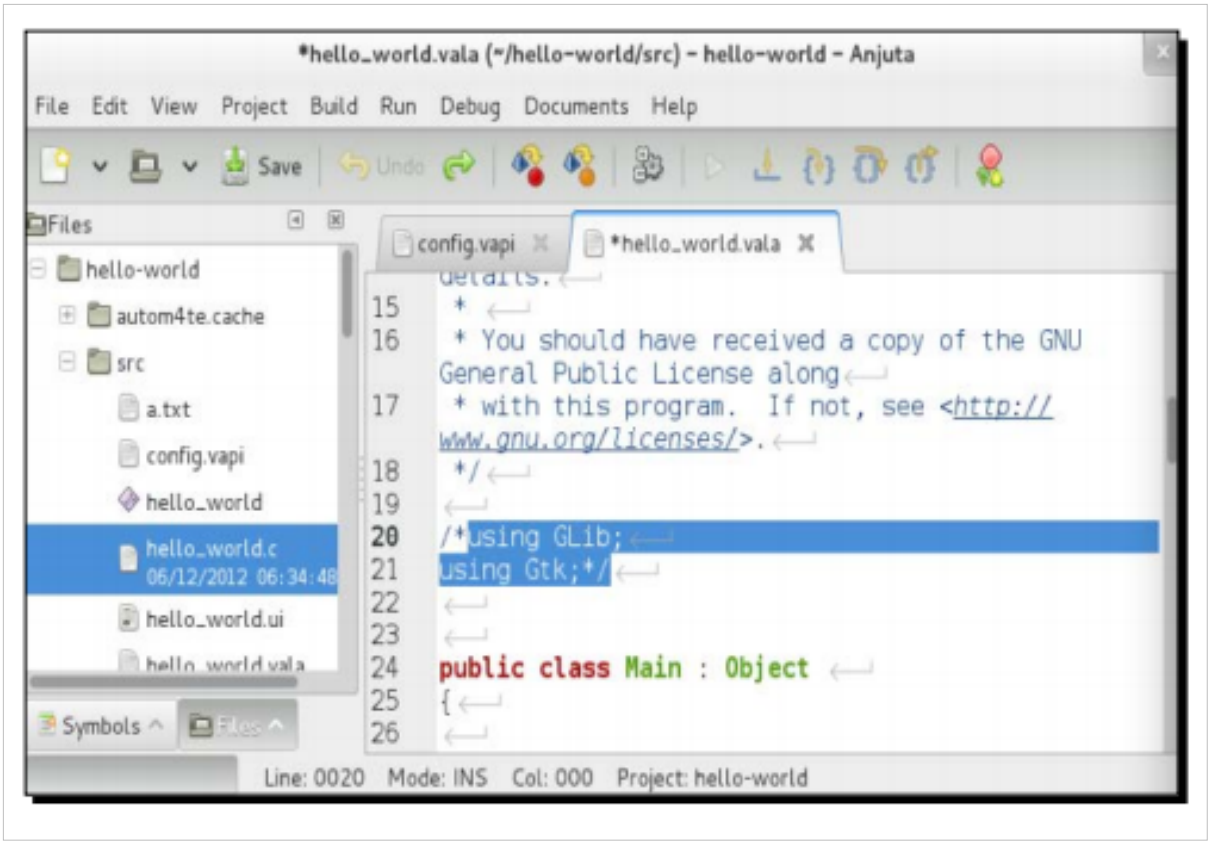
실행하기 - 블록 주석 처리/주석 제거

텍스트 블록을 주석 처리/주석 제거하는 방법을 학습하기 위해서는 아래 과정을 따른다.

1. hello_world.vala 파일을 열거나 앞에서 소개한 활동을 반복하여 파일을 활성화한다.
2. 소스 코드 텍스트에서 아래와 같은 행을 검색하라.

```
using Glib;
using Gtk:
```

3. 마우스 커서를 이용해 두 행을 강조하라.
4. **Edit** 메뉴를 활성화하고 **Comment/Uncomment** 를 선택하라.
5. 텍스트가 주석 처리 될 것이다. 선택된 텍스트를 둘러싼 **/* */** 쌍이 보이는가? 이 명령으로 인해 자동으로 생성된 것이다.



무슨 일이 일어났는가**?**

Anjuta는 텍스트의 선택 블록을 C 스타일의 주석으로 래핑한다. 텍스트 앞에는 /* 문자를, 끝에는 */ 문자를 놓는다. 따라서 후에 Vala와 JavaScript 프로젝트에도 이 방법을 적용할 수 있을 것이란 뜻이다.

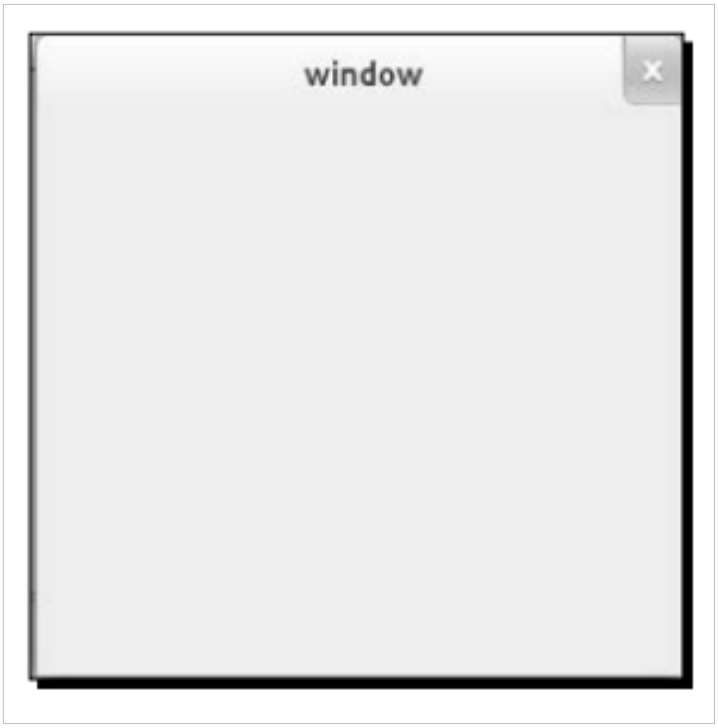
시도해보기 - 블록의 주석 제거하기

앞에서 소개한 단계들을 반복하여 블록의 주석 제거를 시도해보라. 지금 시도하지 않으면 다음에 소개할 예제 활동이 작동하지 않을 것이다!

실행하기 - 처음으로 프로그램 실행하기

아직 프로젝트에서 어떤 것도 변경하지 않았으니 (앞 절에서 텍스트만 주석 제거하였다) 실행해보자.

1. **Run** 메뉴를 클릭하고 **Execute** 를 선택하라.
2. 팝업 대화상자가 열릴 것이다. 어떤 내용도 변경하지 않고 **Execute** 버튼을 클릭하라.
3. Anjuta 가 프로그램을 빌드하는 동안 기다려라.
4. 아래 스크린샷과 같이 화면에 빈 창이 뜬다.



축하한다! 작지만 중요한 단계를 하나 마쳤다!

무슨 일이 일어났는가?

이제 눈에 보이는 것은 어떤 내용도 없는 작은 회색의 창 뿐이다. 내용이 없는 이유는 아직 아무 것도 추가하지 않았기 때문이다. 사실 창 자체는 앞서 시작한 **Create a new project** 마법사에 의해 추가되었다. 마법사는 hello_world.vala 파일 내에 Main이라 불리는 클래스를 생성하였다. 그 안에는 아래와 같은 코드 조각이 보일 것이다.

```
var builder = new Builder ();
builder.add_from_file (UI_FILE);
builder.connect_signals (this);
```

이 코드는 hello-world.ui 파일을 열어 메모리에 로딩한다. 하지만 파일을 로드하기 전에 소스 코드에 아래와 같은 행을 정의한 바 있다.

```
/*
*Uncomment this line when you are done testing and building a tarball
* or installing
*/
// const string UI_FILE = Config.PACKAGE_DATA_DIR + "/" + "hello_
world.ui";
const string UI_FILE = "src/hello_world.ui";
```

이는 앞서 언급한 UI_FILE 파일이 src/hello_world.ui으로부터, 즉 우리의 프로젝트 디렉터리 내부에서 비롯된 것을 의미한다. 여기에 위치한 주석은 의심스럽게 보이지만 사실상 코드 첫 행에 포함된 주석의 제거와 마지막 행의 주석 처리에 필요한데, 배치 중 hello_world.ui 파일은 더 이상 src/ 하위디렉터리에 상주하지 않기 때문이다.

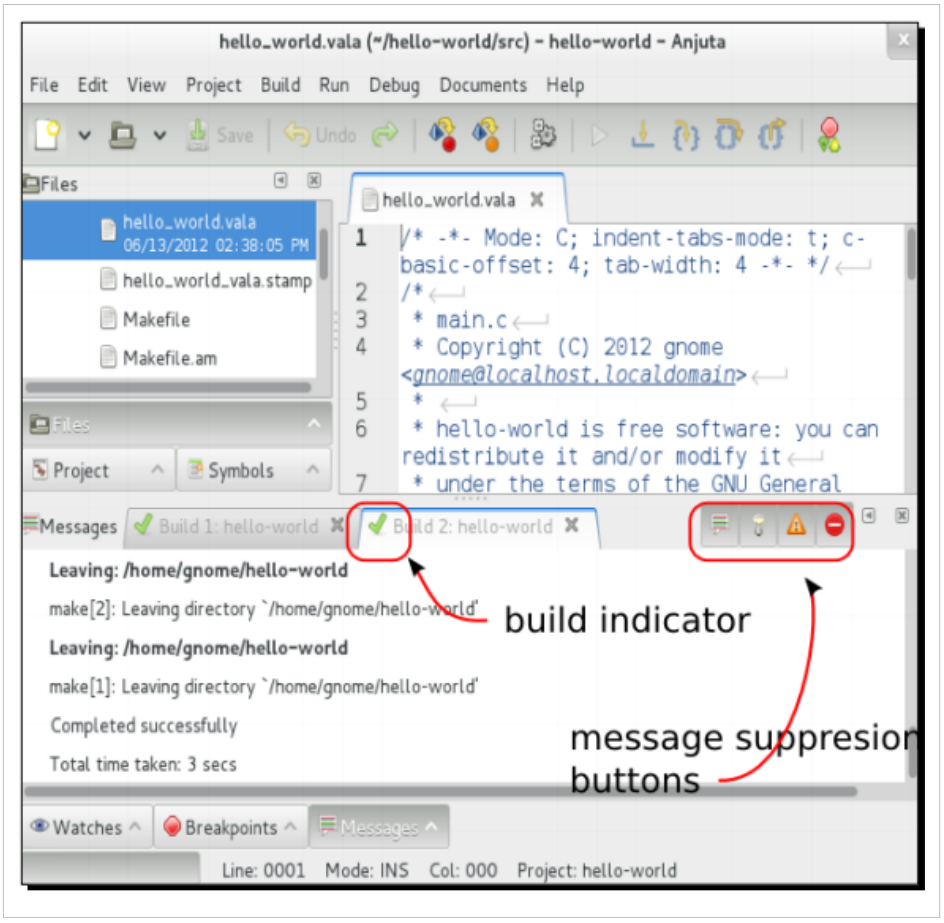
```
var window = builder.get_object ("window" as Window;
window.show_all ();
```

그리고 이 코드는 hello-world.ui 정의로부터 window 인스턴스를 찾아 화면에 표시한다.

여기서 요점은 프로그램이 구분된 파일인 hello-world.ui를 UI 레이아웃 정의로 사용하고 내용을 화면에 표시한다는 점이다. 따라서 위젯을 코드에 수동으로 생성하지 않는단 뜻이다.

프로그램을 실행하기 시작하면 Anjuta는 먼저 프로젝트를 저장하고, 소스 코드를 컴파일함으로써 executable binary를 빌드하여 실행한다. 전체적인 빌드 과정은 **Message** Dock에 표시된다. 우리가 사용 중인 Vala처럼 컴파일된 언어에서는 빌드가 두 가지 단계로 구성되는데, 바로 configure 단계와 make 단계가 그것이다. 두 가지 단계는 **Message** Dock에 탭으로 표시된다.

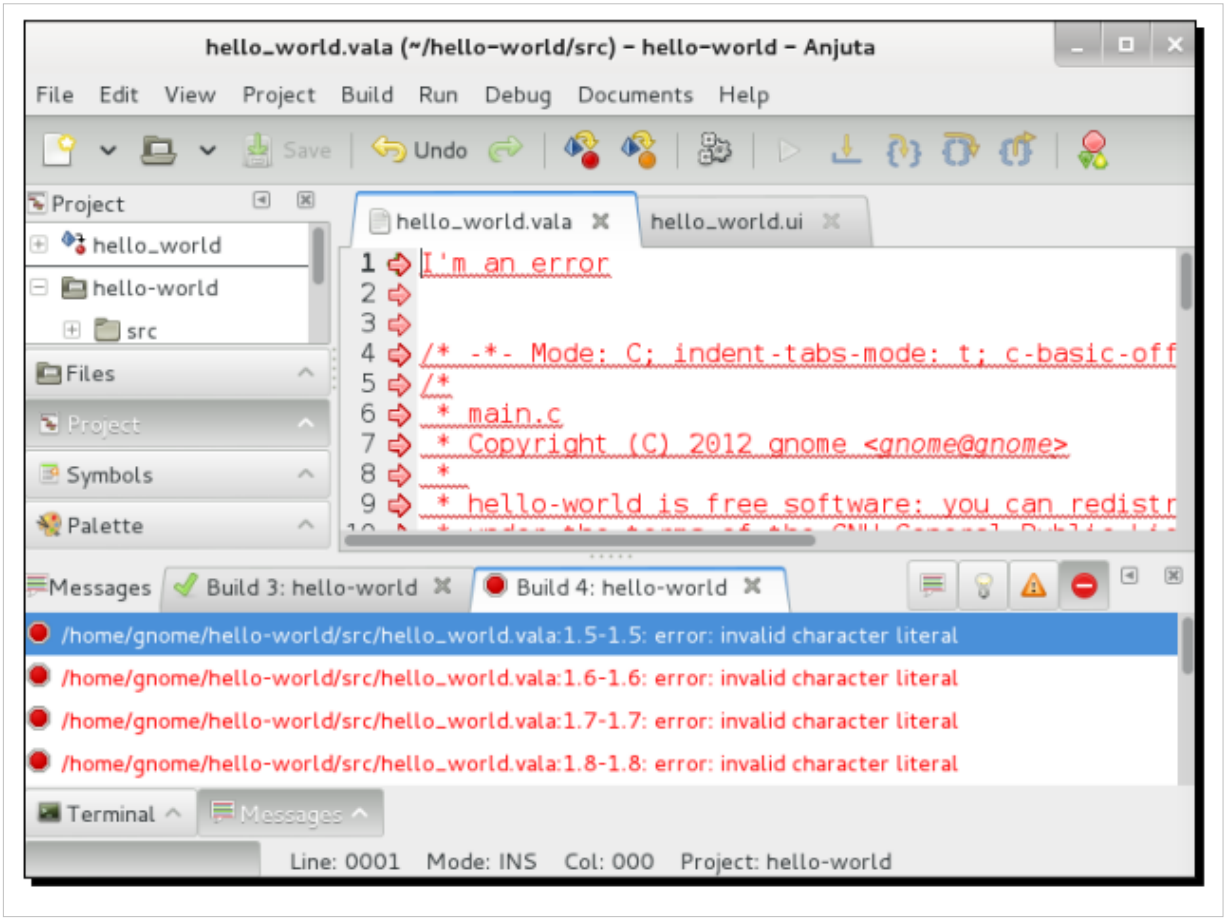
각 단계에서 빌드 상태는 해당하는 빌드 단계 탭에서 아이콘으로 표시된다. 이 아이콘을 살펴보면 빌드의 실패 여부를 확인할 수 있다. 문제가 없이 괜찮으면 녹색 체크 아이콘이 표시된다.



기본적으로 Anjuta는 탭 안에 빌드 과정의 결과를 표시한다. 하지만 메시지 압축 토글 버튼을 누르면 결과를 압축할 수도 있다. 이러한 버튼들은 일반 정보, 경고, 오류 메시지를 표시하거나 숨기는 데에 사용된다. 무엇이 잘못되었는지에 집중하고 빌드 툴로부터 수신되는 다른 메시지 때문에 혼동되지 않도록 오류 메시지와 경고 메시지를 제외한 메시지를 모두 숨기는 편이 유용하기도 하다.

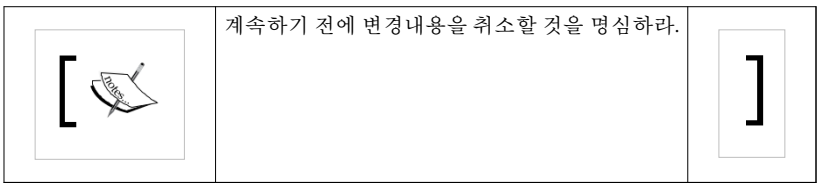
오류를 만들어 어떻게 작용하는지 살펴보기

소스 코드를 수정하되, 가령 hello_world.vala 파일에서 거의 모든 코드 위에 I'm an error를 입력해보도록 하자. 내용을 저장하고 다시 실행해본다. 빌드 표시기 아이콘에 빌드가 표시되었음이 나타날 것이다. 한 번에 하나씩 압축 버튼을 눌러 Anjuta가 표시하는 메시지를 압축해보자. 이러한 버튼들은 불필요한 메시지를 걸러내는 데에 유용하다는 사실을 알아낼 것이다.



이제 오류 메시지를 제외한 모든 메시지를 비활성화시켜보자. 오류가 보이면 더블 클릭해보라.

무슨 일이 일어났는지 보이는가? 그렇다, 에디터는 오류가 발생한 위치를 즉시 표시한다.



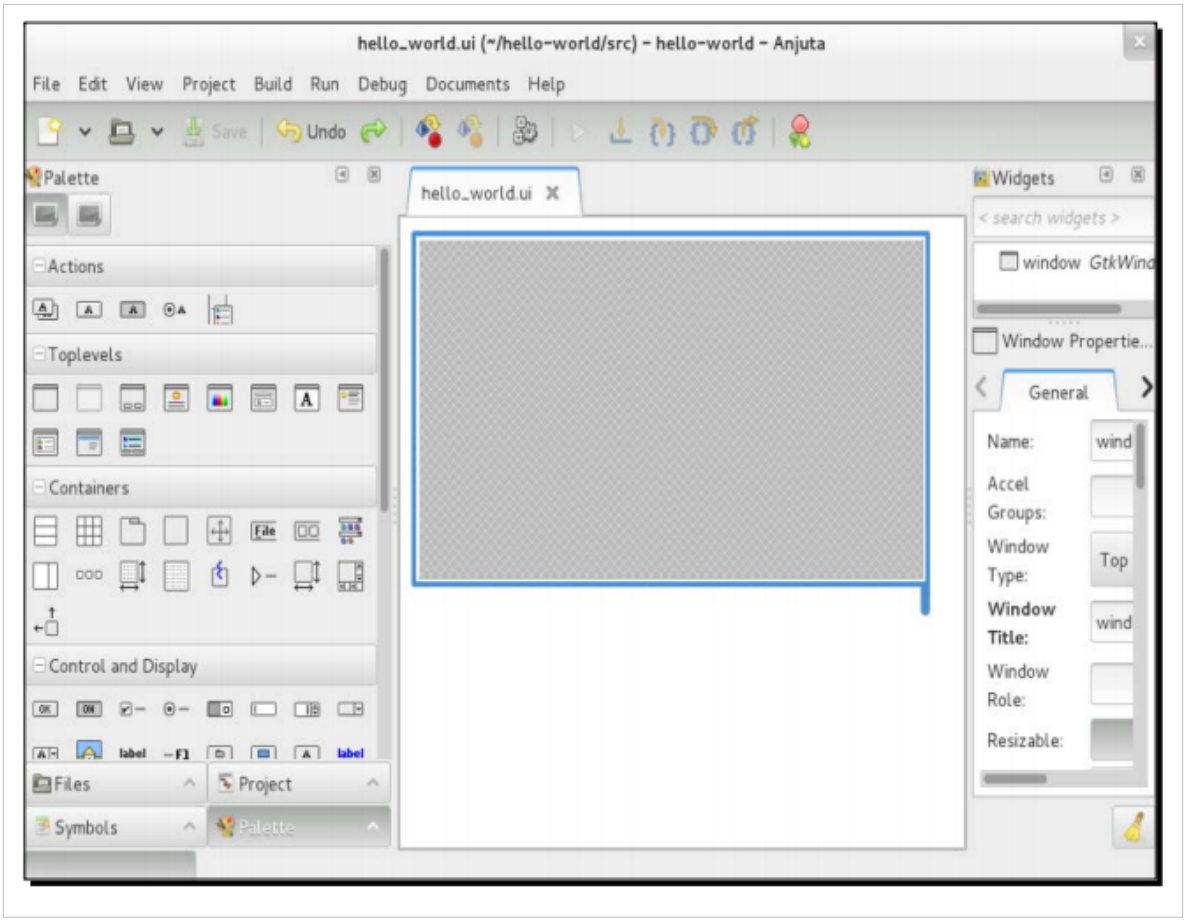
UI 편집하기

Anjuta에서는 소스 코드뿐만 아니라 UI도 편집이 가능하다. 앞서 우리는 내용이 아무 것도 없는 창만 살펴보았다. 이제 진전을 보일 때가 되었다.

실행하기 - UI 편집하기

사용자 인터페이스의 편집을 시도해보자.

1. Project Dock를 활성화하고 hello_world 폴더를 클릭하라.
2. 확장되면 src 폴더를 확장하라.
3. ui 객체를 확장하라.
4. hello_world.ui 항목이 열거되어 있을 것인데, 이를 더블 클릭하라.
5. 사용자 인터페이스 에디터가 열렸다.



[자신의 Linux 배포판에서 hello_world.ui 파일이 GTK+ 버전 2를 필요로 한다는 대화상자를 표시할 경우 염려하지 않아도 된다. 전혀 해로운 것이 아니니 그냥 진행해도 좋은데, 이러한 메시지들은 파일을 저장하면 사라질 것이기 때문이다. 이는 New application wizard가 호출한 초기 코드 생성기가 hello_world.ui 파일에 Version 2 정의를 올바르지 않게 넣었기 때문에 발생한다. 후에 저장하면 이러한 정의는 Version 3으로 대체된다.]

무슨 일이 일어났는가**?**

사용자 인터페이스 에디터는 Glade로, Anjuta 안에 포함되어 있다. 이것은 Anjuta가 완전히 통합되도록 보장하는 바람직한 기능이다. GNOME 시스템 메뉴에서 개별적으로 실행할 경우 정확히 같은 에디터를 발견할 것이다.

Glade는 Anjuta에 두 개의 추가 Dock를 제공한다. 이를 살펴보도록 하자.

팔레트

해당 Dock에는 제 3자 위젯을 포함해 선택 가능한 GTK+ 스톡 위젯이 포함되어 있다. 이러한 이용 가능한 위젯을 선택해 프로젝트 안에 넣을 수 있다. 범주명을 클릭하면 범주 내에서 위젯을 포함하거나 숨길 수 있다. 위젯 아이콘 위에 마우스를 갖다 대면 위젯명이 표시된다.

레이아웃을 변경하지 않았다면 이 Dock는 보통 Anjuta 창의 좌측면에 상주한다.

위젯

이 Dock는 두 가지 부분, 위젯 트리와 위젯 프로퍼티로 나뉜다. 위젯 트리는 위젯의 상속구조를 표시하고, 트리 내 위젯을 클릭하면 선택된 위젯의 프로퍼티가 위젯 프로퍼티에 표시된다. 레이아웃을 변경하지 않은 경우 이 Dock는 보통 Anjuta 창의 우측면에 상주한다.

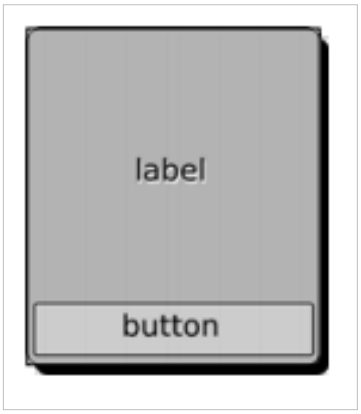
이제 이러한 툴들을 사용해보자.

예제 코드 다운로드하기

계정에서 구매한 모든 Packt 서적의 예제 코드 파일은 http://www.packtpub.com 에서 다운로드할 수 있다. 서적을 다른 곳에서 구매했다면 [1] 를 방문해 등록하면 이메일 주소로 직접 파일을 받을 수 있다.

실행하기 - 라벨과 버튼 추가하기

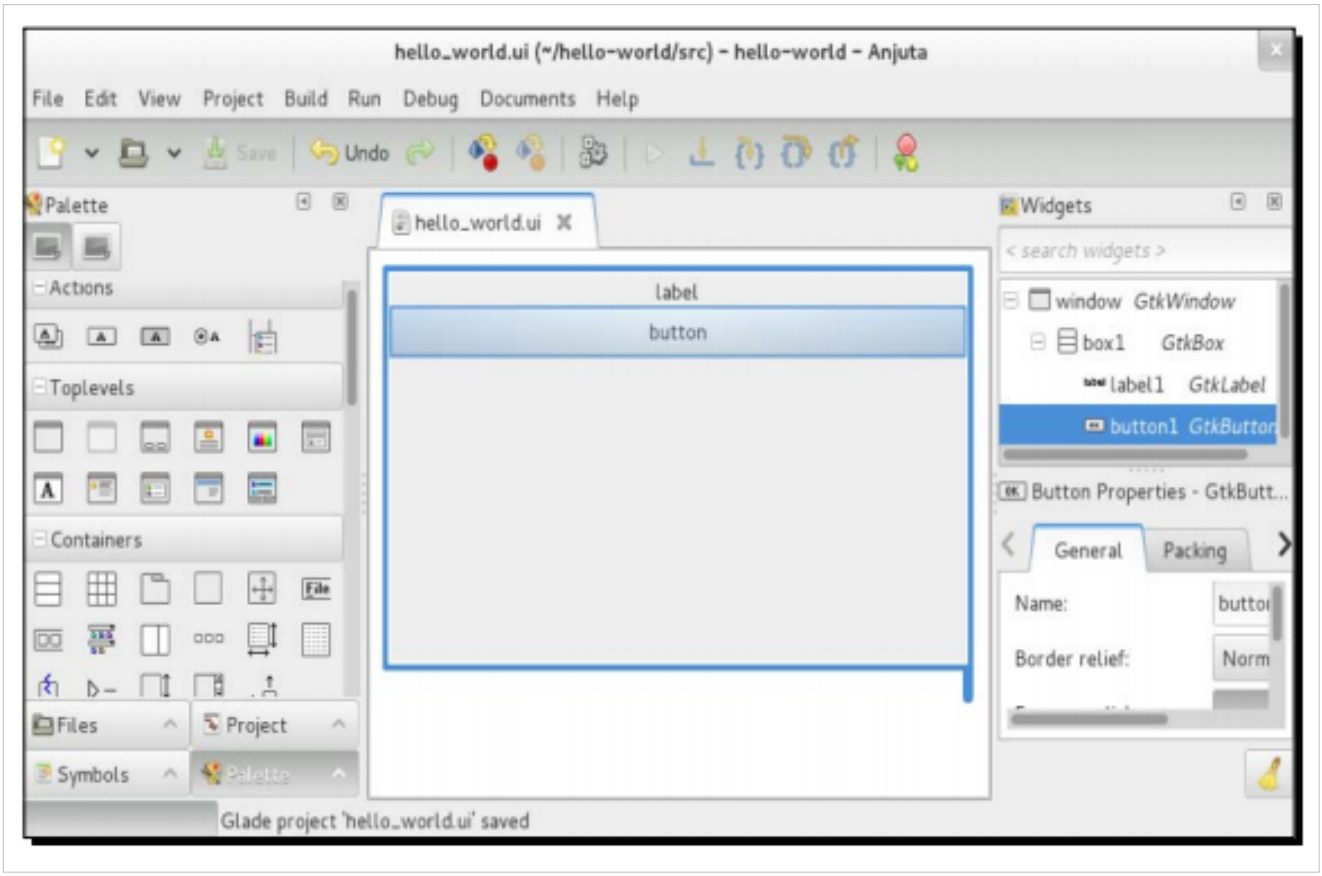
UI 레이아웃을 편집하면서 UI mockup을 지침서로 따를 예정이다. 목업(Mockup)은 UI의 대략적인 디자인으로, 애플리케이션 디자이너들이 주로 생성한다. 이러한 목업은 이후 구현할 개발 팀으로 제시된다. 이제 목업을 고려해보자.



목업을 바탕으로 두 개의 위젯, 즉 라벨과 버튼이 필요함을 볼 수 있다. 라벨은 버튼의 상단에 배치된다. 이제 이 위젯들을 창으로 놓을 것이다. 아래 단계를 실행하라.

1. **Palette** Dock에서 **Container** 범주를 찾아라.
2. 박스 아이콘을 클릭하여 **Box** 를 선택하라 (아이콘 리스트 위에 마우스를 갖다 대면 **Box** 객체를 찾을 수 있음을 기억하라).
3. 중앙에서 Anjuta가 생성한 **window** 객체를 클릭하라.
4. 대화상자가 나타나면서 박스 안에 얼마나 많은 항목을 유지할 것인지 물을 것이다. 라벨과 버튼에 2 를 입력하라.

5. 박스는 **window** 객체에서 삽입될 것이며, 거의 눈에 띄지 않겠지만 창이 이제 두 부분으로 나뉘었음을 확인할 수 있을 것이다.
6. 우측에 **Box Properties** 를 검사하라. **Orientation** 옵션을 **Vertical** 값으로 되게 하고, 값이 **Horizontal** 일 경우 변경하라.
7. 목업에서 라벨은 상단에, 버튼은 하단에 있음을 확인할 수 있다.
8. **Palette** 의 **Control and Display** 범주에서 **label** 위젯을 찾아 클릭하라.
9. 중앙에 박스 상단 부분을 클릭하라. 라벨의 크기에 따라 상단 부분이 줄어드는 것을 확인할 것이다. 그대로 두어라.
10. **Palette** 의 **Control and Display** 범주에서 **button** 위젯을 찾아 클릭하라.
11. 박스에서 이제 유일하게 이용할 수 있는 하단 부분을 클릭하라.
12. 모든 위젯이 제자리에 위치하게 되었다!



무슨 일이 일어났는가**?**

위젯을 성공적으로 창에 추가하였다. 안타깝게도 버튼과 라벨을 직접 창으로 추가할 수는 없는데, 이는 GTK+가 작동하는 방식이 아니기 때문이다. 창은 하나의 위젯만 포함할 수 있다. 따라서 우리는 하나 이상의 위젯을 보유할 수 있는 컨테이너가 필요하다. 그래서 버튼과 라벨을 추가하기 전에 컨테이너를 추가해야 한다.

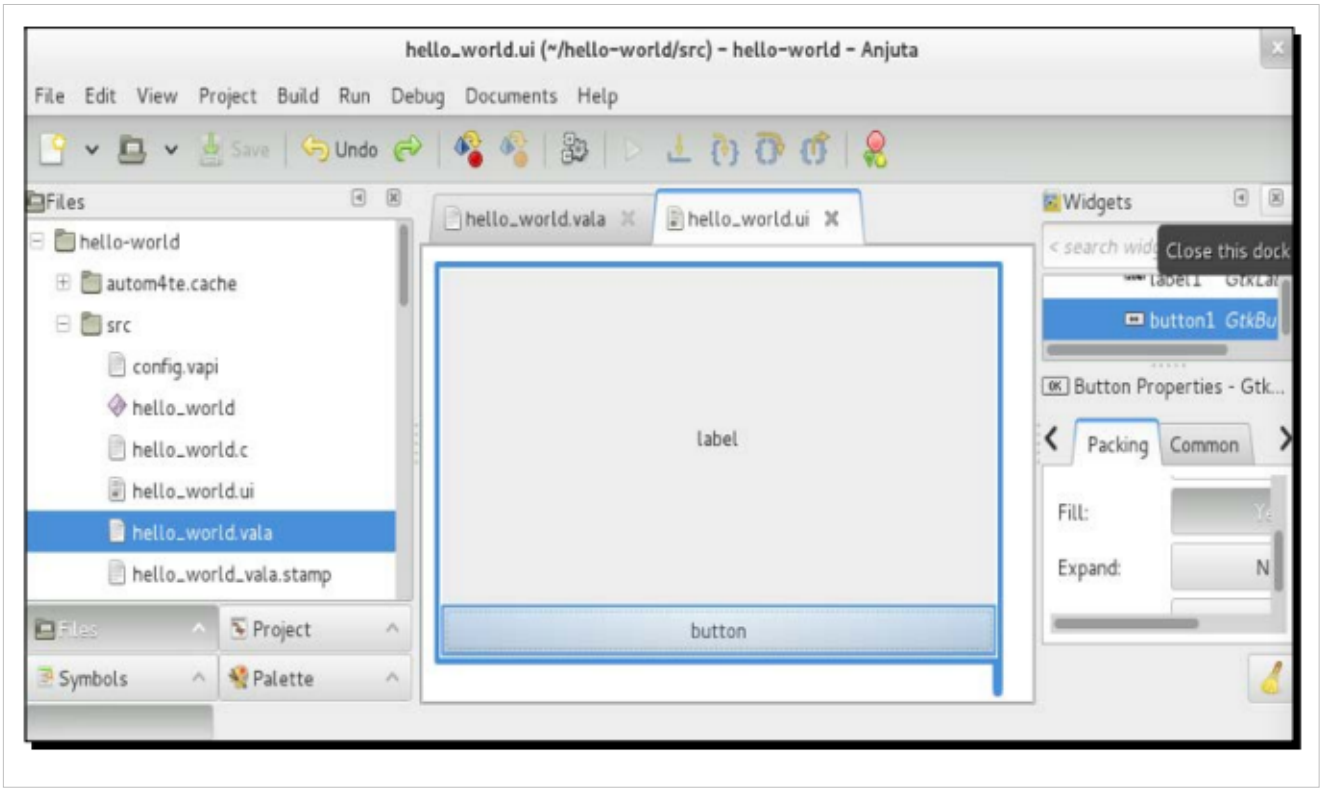
상속구조의 이해는 GTK+ 애플리케이션을 개발 시 매우 중요하다. 우측에 **Widgets** Dock의 상단 부분을 살펴보면 상속구조에 위젯이 어떻게 표시되는지 확인할 수 있다. 젤 위에는 **window** 객체가 있다. **window** 객체에는 **box1** 이라는 자식이 있다. **box1** 에는 두 개의 자식, **label1** 과 **button1** 이 있다. 위젯은 좌표체계 내 특정 위치에 놓이지 않고, 상속구조를 따라 하나씩 패킹된다.

이제 어떤 모습인지 프로그램을 실행해보자. 버튼을 클릭할 수 있도록 하라.

실행하기 - 위젯 프로퍼티 변경하기

확인할 수 있듯이 마지막 활동의 결과는 만족스럽지 못하다. 목업 디자인에서 예상한 결과가 아니다. 이를 해결하도록 조치를 취해보자.

1. 박스의 상단 부분에 **label** 위젯을 클릭하라. 라벨이 강조되었음을 주목하라.
2. **Widgets** dock 의 **Label Properties** 에서 **Packing** 탭을 클릭하라.
3. **Expand** 옵션을 찾아 **Yes** 를 선택하라.
4. 드디어 우리 목업과 같은 모습이 되었다.



반응(responsive) 버튼

이 시점에서 우리 프로그램은 스스로를 표시하는 것 외에는 어떤 일도 수행하지 않는다. 이제 버튼이 클릭 이벤트에 응답하도록 변경해보자. 예를 들어, 버튼을 클릭하면 무언가를 인쇄하길 원한다고 가정해보자.

실행하기 - 반응하는 버튼으로 만들기

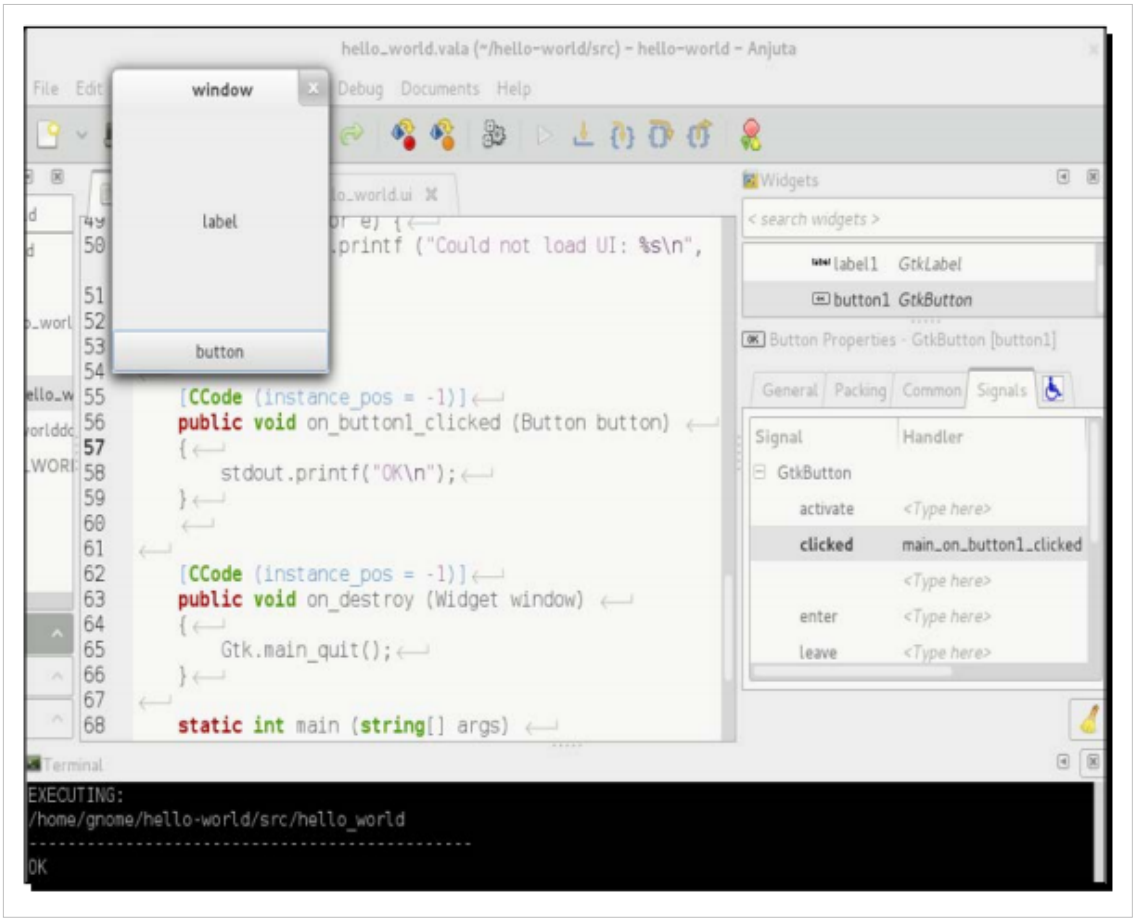
이제 이를 실현하기 위해 코드를 조금 추가해보자.

1. 앞서 생성한 **button** 객체를 클릭해 강조해보자.
2. **Button Properties** 를 체크하고 그 안에서 **Signals** 탭을 찾아라. 찾을 수 없다면 탭의 우측에 **>** 버튼을 클릭하라. **Signals** 를 찾을 때까지 계속 클릭하라.
3. **button** 은 기본적으로 Gtk 버튼이므로 리스트 상에서 **GtkButton** 엔트리를 확장시켜라. 행과 열의 수가 거의 없다. 리스트에서 **clicked** 시그널을 찾아라.
4. **handler** 열에서 **clicked** 시그널을 클릭하여 텍스트 엔트리가 활성화되도록 하라. 엔트리에 main_on_button1_clicked 를 입력하라.
5. hello_world.vala 의 소스 코드로 돌아가 Main 클래스의 본체(body) 내부를 살펴보라. 해당 코드를 클래스 범위 내 어느 곳이든 추가해도 좋고, 아니면 Main 클래스 내에 public void on_destroy 메서드를 찾아 그 위에 아래 코드를 추가하는 방법을 이용해도 좋다.

```
[CCode (instance_pos = -1)]
public void on_button1_clicked(Button button) {
    stdout.printf("OK\n");
```

```
}
```

6. 이제 프로그램을 실행하고 버튼을 클릭하라.
7. 버튼을 클릭하면 **Terminal** Dock 에 **OK** 가 출력됨을 확인할 것이다.



무슨 일이 일어났는가**?**

소스 코드에 쓰인 시그널 핸들러와 Glade에서 생성된 **button1** 위젯에 명명된 시그널이 연결되었다. 우리가 연결한 시그널명은 clicked로, 버튼을 클릭할 때마다 **Handler** 열에 명시된 시그널 핸들러가 호출된다. 앞서 hello_world.vala에 작성한 함수와 상관된 main_on_button1_clicked를 **Handler** 열에 명시한다.

우리는 핸들러 함수를 on_button1_clicked로 명명하고 Main이라는 클래스에 넣었다. 시그널 핸들러명에 main_on_button1_clicked라는 이름을 이용한 것도 이 때문이다. 이 때는 한 가지 규칙이 표시되는데 소문자로 작성된 클래스명과 함수명을 결합하되 밑줄로 구분해야 한다는 것이다. 함수명 자체는 이벤트가 발생했다는 인상을 주어야 한다.

그렇다면 hello_world.vala 파일의 검사 내용을 백업해보자. 아래와 같은 코드를 갖고 있을 것이다.

```
var builder = new Builder ();
builder.add_from_file (UI_FILE);
builder.connect_signals (this);
```

마지막 행이 이제 이해가 된다. builder객체 내의 connect_signals 메서드가 시그널 연결을 책임진다는 말이다. 이 행이 없이는 핸들러 함수의 쓰기와 앞의(previous) 지침서를 이용하더라도 전체 프로그램이 작동하지 않을 것이다. 이 행은 기본적으로 사용자 인터페이스 위젯이 발생시킨 모든 시그널을 이 객체, 즉 Main 객체로 묶는다. 그래서 핸들러 함수를 Main 안에 넣는 것이다.

팝 퀴즈 - 시그널 명명하기

훌륭한 프로그래밍 실제는 우리의 삶을 매우 쉽게 만든다. 그러한 실제 중 하나는 규칙을 고수하는 것에 해당한다. 규칙을 고수할 때 한 가지 장점은 디버깅 시 우리의 마음이 코드를 직접적으로 추적할 수 있다는 것이다. 규칙은 조직마다, 심지어 프로젝트마다 다양할 수 있다. 개발자는 특히 프로젝트에서 하나의 규칙을 고수해야 하며, 필요 시 다른 프로젝트에서는 다른 규칙으로 전환할 수 있다.

연결이 시작되면 이벤트를 처리하는 함수와 Server 클래스가 있다고 상상해보라.

Q1. 앞서 논한 규칙을 바탕으로 했을 때 함수에 부여하는 이름으로 어떤 것이 가장 좋을까?

1. server_on_connection_started.
2. server_connection.
3. start_connecting.

기호 추적하기

변수, 메서드, 상수 또는 그와 비슷한 것인 경우 기호는 Anjuta에서 쉽게 추적이 가능하다. 앞서 논한 UI_FILE 상수를 기억하는가? 소스 코드 파일의 깊은 곳에서 아래와 같은 코드 조각을 우연히 발견했다고 상상해보자.

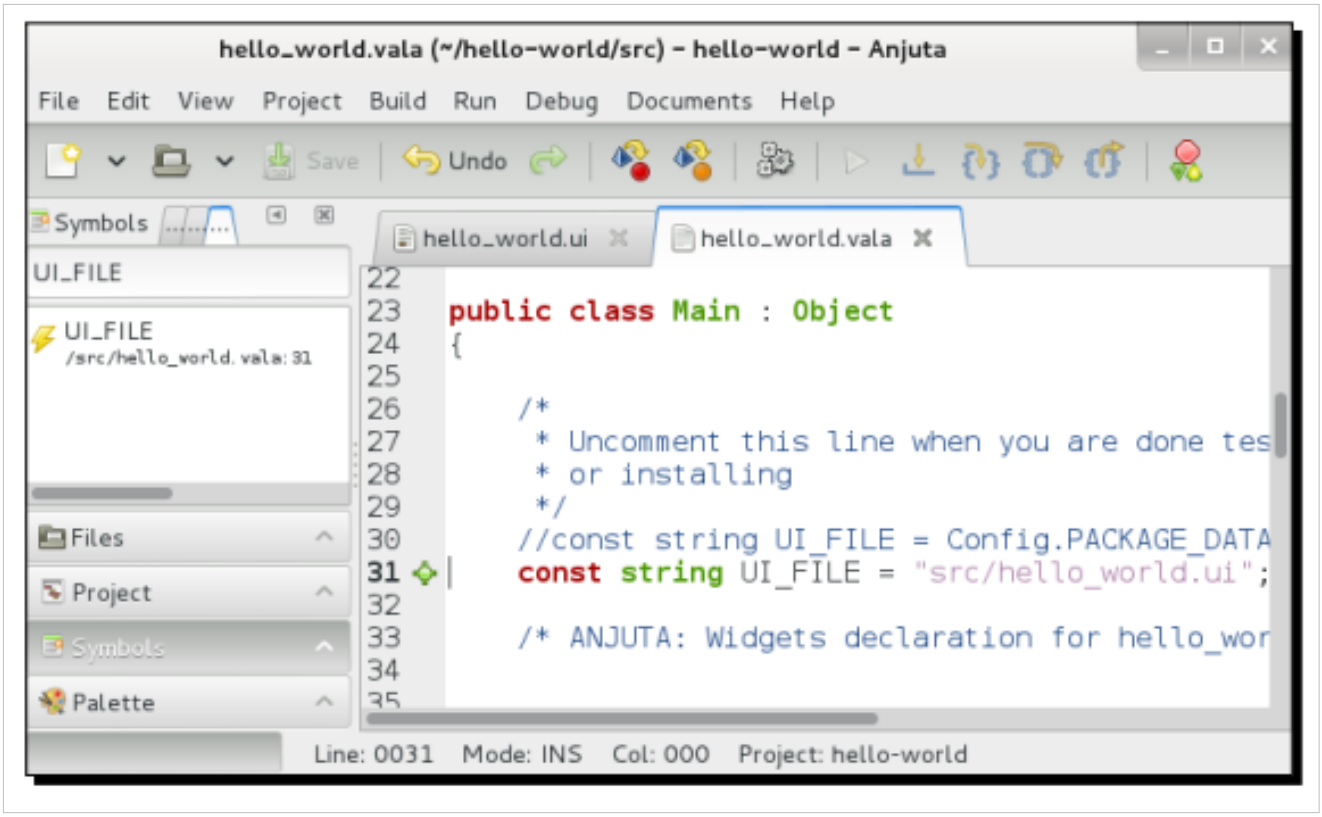
```
builder.add_from_file (UI_FILE);
```

UI_FILE 의 값이 무엇인지, 누가 정의했는지 등이 궁금할 것이다.

실행하기 - 기호 찾기

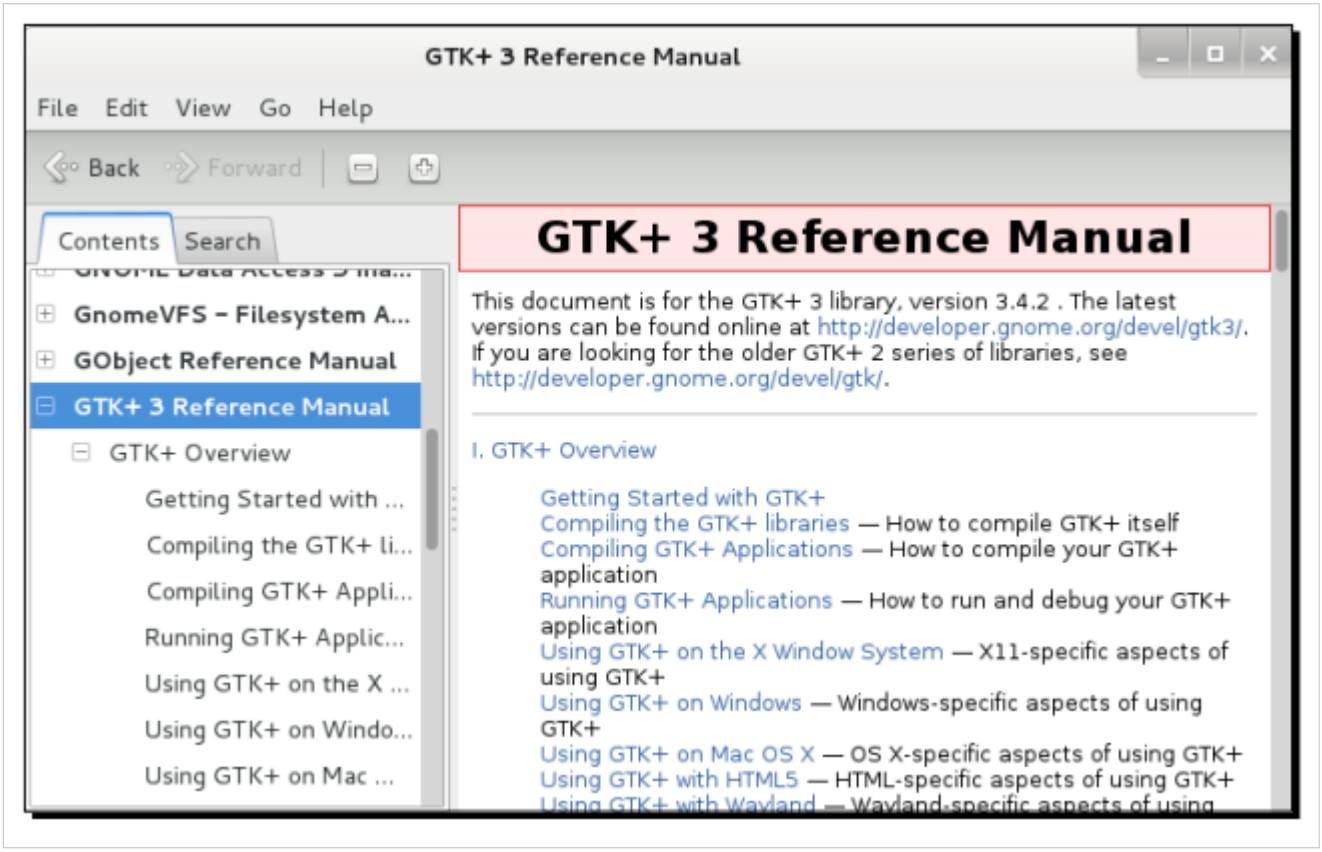
찾는 방법을 연습해보자.

1. **Symbols** Dock를 클릭하여 활성화하고, 화면에 Dock가 보이지 않으면 **View** 메뉴로 가서 활성화하라.
2. UI_FILE이 열거되어 있음을 확인할 것이다. 하지만 프로젝트가 너무 커서 그 안에 상수가 묻혀 있다고 생각해보자. Dock에 **Search** 탭으로 가보자.
3. 탭에 UI_FILE을 입력해보라. 오타를 피하도록 훌륭한 코드 완성 기능이 있음을 주목하라.
4. UI_FILE은 그것이 정의된 장소와 함께 리스트에 표시된다.
5. 검색 결과를 더블 클릭하라. 에디터는 정확히 그것이 어디에 정의되어 있는지 보여줄 것이다. 깔끔하지 않은가?


hello_world.vala (~/hello-world/src) - hello-world - Anjuta
File Edit View Project Build Run Debug Documents Help
Save Undo
Symbols
hello_world.ui
hello_world.vala
UI_FILE
UI_FILE
/src/hello_world.vala: 31
Files
Project
Symbols
Palette
22
23 public class Main : Object
24 {
25
26 /*
27 * Uncomment this line when you are done tes
28 * or installing
29 */
30 //const string UI_FILE = Config.PACKAGE_DATA
31 const string UI_FILE = "src/hello_world.ui";
32
33 /* ANJUTA: Widgets declaration for hello_wor
34
Line: 0031 Mode: INS Col: 000 Project: hello-world

도움 얻기

소프트웨어를 개발 시 도움을 얻기 위해서는 양호한 참조 툴이 필요하다. 우리는 Devhelp라는 툴을 논하고자 한다. 이것은 오프라인 **Application Programming Interface(API)** 참조 툴로, 간단하고 빠르기 때문에 이용하고자 한다. 해당 툴은 읽고 검색이 가능한 매뉴얼 모음("books"라고 불림)을 연다.



사용법은 꽤 단순하다. 좌측에는 책이 표시된다. 책을 읽기 위해서는 우측의 페이지 본문에 표시된 링크를 따라가거나 단순히 확대하여 빠르게 훑어볼 수도 있다.

요약

지금까지 Anjuta에서 마법사를 이용해 Vala 프로젝트를 구현함으로써 Vala 프로그래밍을 소개해봤다. IDE 레이아웃은 사용하기가 꽤 쉽다는 사실을 발견했으며, 보고 싶지 않은 요소는 숨기거나 표시할 수 있음을 알아냈다. 또 애플리케이션을 빌드하는 단계와, 빌드가 실패할 때 추가 정보를 찾는 방법도 이해하게 되었다. 빌드 과정에서 무언가 잘못되면 혼동을 피하도록 메시지를 걸러낼 수도 있다.

소스 코드의 탐색은 Anjuta에서 쉽게 실행이 가능하다. Anjuta는 코드를 조작하도록 몇 가지 단축키를 제공하기 (예: 코드 블록의 주석 처리와 주석 제거) 때문에 비교적 쉽게 가능하다 .

Glade를 이용해 목업 디자인을 기반으로 UI를 구현할 수 있었다. 또 Glade는 Anjuta로 깊숙이 통합됨을 알고, Glade를 구분된 애플리케이션으로 실행할 수도 있음을 학습했다. Glade에서 UI가 어떻게 만들어지는지, 프로그램이 어떻게 읽고 사용하는지, UI 객체의 시그널을 핸들러 코드로 연결함으로써 어떻게 작동하는지도 살펴보았다. 개발 중에 도움이 필요하면 API 참조 문서로 접근하도록 해주는 Devhelp를 이용할 수 있다.

이제 GNOME 애플리케이션 개발을 학습할 준비가 되었을 것이다. 필요한 기술은 모두 준비되었다. Vala의 소개를 끝마치고 이제 JavaScript와 함께 GNOME 애플리케이션 개발 단계를 하나 더 나아가 심도 있게 학습하도록 하자.

Notes

GNOME3ApplicationDevelopmentBeginnersGuide:Chap 03

제 3 장 프로그래밍 언어

프로그래밍 언어

*GNOME3*가 생기기 오래 전에는 *C*가 *GNOME* 애플리케이션을 생성하는 첫 프로그래밍 언어였는데, 이후로 *C++, C#, Python,* 그 외의 언어들이 생겨났다. *GNOME*이 버전 *3*에 가까이 발전하자 *Vala*와 *JavaScript*는 더욱 더 유명해지고 중요한 부분이 되었다. *JavaScript*는 오랜 시간 사용되었고 사람들도 이에 익숙하다. *Vala*는 비교적 새로운 언어에 해당하지만 이것으로 작성한 프로그램은 속도가 빠르고, *Java*나 *C#*에서 취하는 구문과 비슷한 구문을 가진다는 이유로 *GNOME* 애플리케이션 개발자들 사이에서 유명세를 얻고 있다.

이번 장에서는 두 개의 프로그래밍 언어를 논하겠다. 먼저 JavaScript와 Vala의 기본 내용을 빠르게 살펴보고 이 언어들을 이용해 애플리케이션을 만들기에 충분한 지식을 습득하도록 하자.

본 장에서 학습할 내용은 다음과 같다.

- JavaScript에서 데이터 유형 다루기
- JavaScript에서 반복 제어하기
- JavaScript에서 기본적인 객체 지향 프로그래밍
- JavaScript 객체 구성하기
- JavaScript 프로토타입 사용하기
- JavaScript 프로그램 모듈화(modularize)하기
- Vala member 접근 명시자
- Vala에서 기본 데이터 유형
- Gee 컬렉션 라이브러리

이제 Seed를 이용해 JavaScript 구현을 먼저 시작해보자.

JavaScript를 이용한 GNOME 프로그래밍

사실 GNOME에는 두 가지 상충되는 JavaScript 구현이 있는데, 둘의 차이는 사용되는 엔진에 있다. 첫 번째 구현은 Gjs로, Mozilla가 생성한 JavaScript 엔진인 Spidermonkey를 기반으로 한다. 두 번째는 우리가 살펴볼 Seed다. 이는 WebKit의 JavaScript 코어 엔진을 기반으로 한다. Seed를 선택한 이유는 공식적으로 GNOME3에서 사용되기 때문이다.

실행하기 - Seed에 인사하기

이제 Seed가 어떻게 작동하는지 살펴볼 때가 되었다.

1. GNOME Shell의 **Activities** 에 위치한 **Terminal** 메뉴 혹은 **Applications | Accessories | Terminal** 으로부터 terminal을 실행하라.
2. Terminal 콘솔에 seed를 입력하여 Seed를 실행하라.

```
$ seed
```

3. Seed 프롬프트를 입력하고 있다.

```
>
```

4. 아래 코드를 입력하고 리턴 키를 입력한다.

```
print("Hello, world")
```

5. 텍스트가 출력된다.

```
Hello, world
```

무슨 일이 일어났는가?

Seed는 인터프리터다. 이렇게 Seed를 실행할 때는 Seed 상호작용 모드를 입력하고 있으므로 우리가 입력한 코드의 결과를 상호작용적으로 제공함을 의미한다. 이 모드에서는 해당 셸에 유효한 JavaScript 코드는 무엇이든 입력할 수 있다. 하지만 JavaScript 프로그래밍에 익숙한 사람들은 웹 애플리케이션을 위한 코딩 시 이와 같은 코드를 입력할 수 없다. 가령 아래의 예제 코드는 입력할 수 없을 것이다.

```
console.log("Hello, world")
```

그 이유는 Seed가 document 또는 console과 같은 객체들을 제공하지 않기 때문이다. 따라서 필요한 객체를 가져오지 않는 한 기본적인 JavaScript만 허용된다.

시도해보기 - 좀 더 많은 JavaScript 시도하기

말이 나온 김에 덧셈, 뺄셈, 변수 할당과 같은 JavaScript 코드를 셸에 입력해보자. 아래를 예로 들어보겠다.

```
var a=1
var b=2
b+a
a-b
```

리스트는 계속된다. 이 리스트는 셸의 기본 개념을 이해하기 위해 제시하였다. JavaScript의 경험이 있지만 기술이 아직 미숙하다면 시작 단계로 이용해도 좋다. 아니면 좀 더 현명하게 이것을 계산기로 사용하는 방도를 고려해도 좋다.

끝내지 않은 행을 준다고 가정하자.

```
var c=
```

프롬프트가 아래와 같이 변하는 것을 볼 것이다.

```
. .
```

이는 우리가 프롬프트를 끝낼 필요가 있으며, 그렇지 않으면 Seed가 구문 오류 메시지를 방출(spew)할 것이란 뜻이다.

이것이 끝나면 Ctrl+C 키 조합을 눌러 꺼도 되는데, 이럴 경우 시스템의 terminal 콘솔로 돌아갈 것이다.

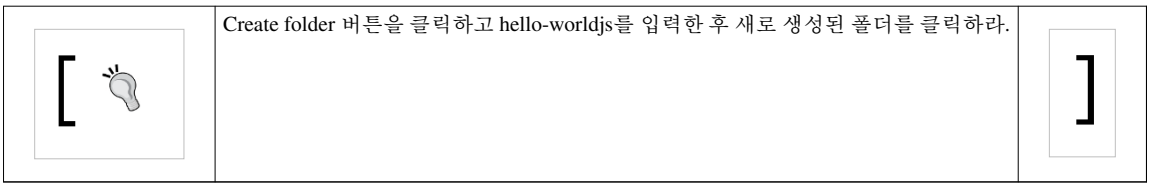
실행하기 - Seed로 프로그램 실행하기

상호작용 모드는 실제 애플리케이션에서는 이용할 법한 접근법이 아니다. 지금부터는 코드를 파일에 넣은 후에 실행하고자 한다. 준비 되었는가?

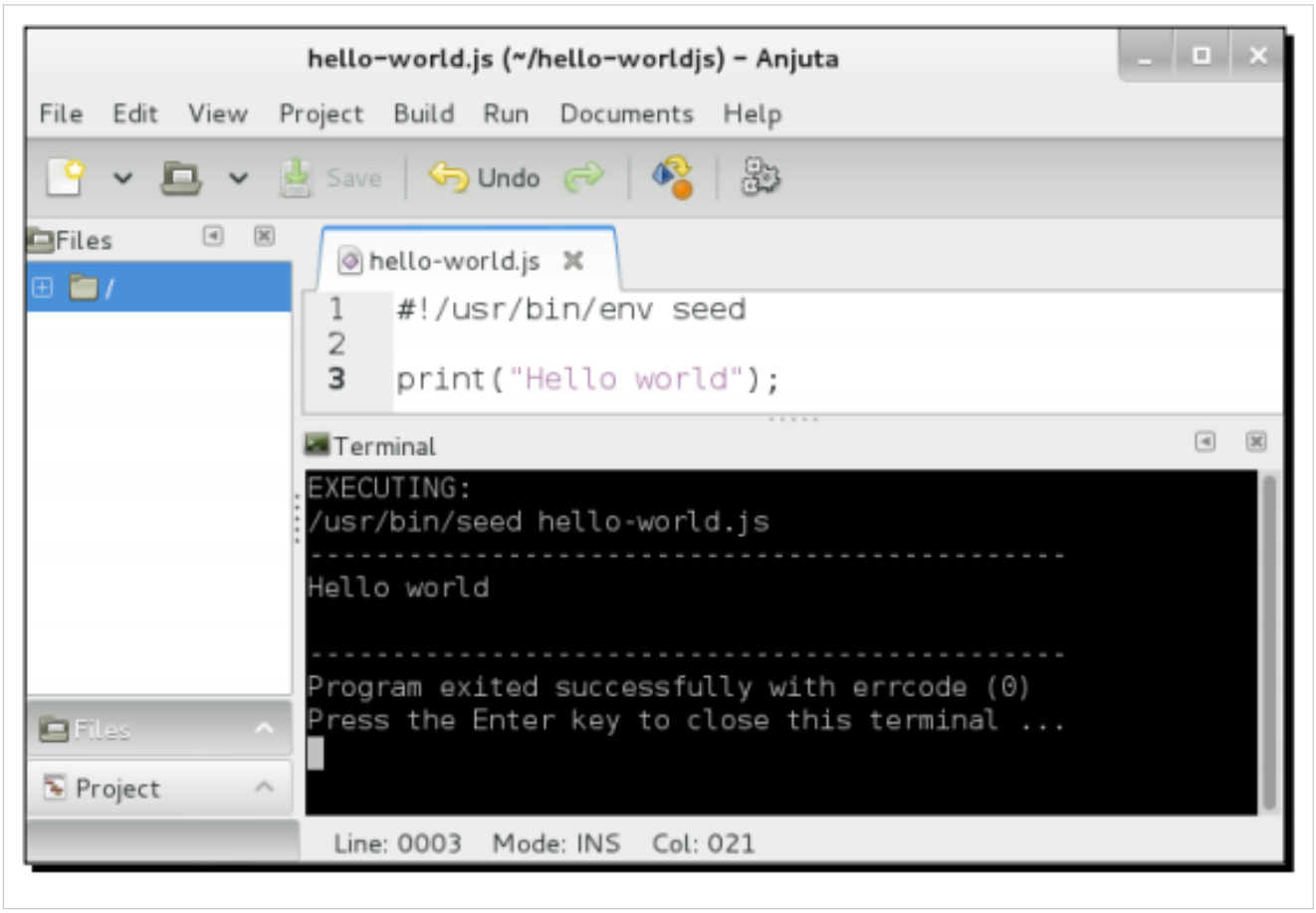
1. Anjuta를 작동시켜라.
2. File | New를 통해 새로운 파일을 생성하라.
3. 아래 코드 조각을 이용해 에디터를 채워라.

```
#!/usr/bin/env seed
print("Hello, world")
```

4. 이 파일을 hello-world.js로 저장하라. 이를 위한 새 디렉터리를 생성하고(예: hello-worldjs) 그 안에 해당 파일을 넣어라.



5. Run 메뉴를 클릭하고 Execute를 선택하라. 작은 대화상자가 표시되면 Program 필드는 /usr/bin/seed로 채우고 Arguments 필드는 hello-world.js로 채운다. Run in terminal 옵션이 체크되어 있도록 하라.



무슨 일이 일어났는가?

이러한 프로그램 호출 방법을 스크립팅이라고 부른다. 해당 접근법을 이용하면 파일 자체가 직접 Seed에 의해 로딩되고 실행된다. 이것은 Bash, Perl, Python과 같은 다른 스크립트와 비슷한 방법을 이용한다. 첫 번째 행에서 스크립트의 인터프리터로 사용된 프로그램을 표시하기 위해 해시 뱅(#!) 기호를 사용함을 확인할 수 있을 것이다. 우리는 /usr/bin/seed를 직접 넣는 대신 /usr/bin/env 다음에 seed를 사용했다. 그 이유는 seed의 위치를 엄격하게 고정하고 싶지 않기 때문이다. env를 이용해 시스템은 Seed의 정확한 위치를 찾도록 시스템 경로를 받아들일 것이다. 예를 들어, /usr/bin 대신 /usr/local/bin에 seed가 있으면 프로그램은 여전히 작동할 것이란 뜻이다.

그렇다면 hello-world.js를 직접 입력하지 않고 여전히 Run 대화상자에 /usr/bin/seed를 넣어야 하는 이유가 궁금할 것이다. 그것은 스크립트의 실행 가능 프로퍼티를 설정하지 않았기 때문이다. Linux 관리 기술을 발휘하여 hello-world.js 를 위치시킨 디렉터리로 가서 terminal에서 아래 명령을 호출해보자.

```
chmod +x hello-world.js
```

그 다음 Run 대화상자의 Program 필드에 직접 hello-world.js를 넣어보자. Run 메뉴와 Execute메뉴를 접근할 때 더 이상 이 대화상자가 보이지 않을 것이다. Anjuta는 우리가 이미 프로그램 인자를 설정하였고 실행할 준비가 되었다고 생각하기 때문이다. 이를 다시 변경하고 싶다면 Run 메뉴로 돌아가 Program Parameters... 메뉴를 선택할 수 있다. 이 방법이 아니라면 아래와 같이 시스템 콘솔에서 직접 스크립트를 실행할 수도 있다.

```
./hello-world.js
```

느슨한 타입의 언어

JavaScript는 느슨한 타입의 프로그래밍 언어로 알려져 있다. 즉, 변수가 숫자든, 문자열이든, 배열이든 그 타입을 선언하지 않고도 사용할 수 있다는 뜻이다. 단순히 변수를 선언하기 위해 var 지시어를 이용하면(혹은 이용하지 않으면) 된다. 어떻게 작동하는지는 곧 살펴볼 것이다.

실행하기 - 데이터 유형 다루기

이제 JavaScript에서 기본적인 데이터 유형을 논하고 어떻게 상호작용할 수 있는지 살펴보자. 이것이 끝나면 필요에 따라 사용할 유형을 선택할 수 있을 것이다.

1. hello-world-data-types.js라고 불리는 새로운 파일을 생성하여 아래 내용으로 채운다.

```
#!/usr/bin/env seed

print("Hello world")

var number = 1;
print(number);
number = number + 0.5;
print(number);
print(number.length);
number = number + " is a number? no, it is now a string";
print(number.length);
print(number);
number = (number.length == 0)
print(number);
number = undefined
print(number);
```

2. 파일을 실행하라.
3. number 변수가 출력되므로, 다음과 같다:

```
Hello world
1
1.5
[undefined]
1.5 is a number? no, it is now a string
39
0
[undefined]
```

무슨 일이 일어났는가**?**

두 가지 흥미로운 점이 눈에 띌 것이다. 첫째, JavaScript는 데이터 유형 간 효율적으로 이용할 수 있다. 새로운 값을 할당하면 하나의 변수 타입에서 다른 타입으로 변경이 가능하다. 두 번째는 지금까지 말했던 바와 같이 변수의 타입을 선언할 필요가 없다는 것이다. 이 코드에는 초기에 값을 1로 설정한 number 변수가 있다.

```
var number = 1;
print(number);
```

이제 number 변수는 일반 정수에 불과하다.

```
number = number + 0.5;
print(number);
```

그리고 0.5를 더하여 부동 소수점 데이터 유형으로 만든다. JavaScript는 문제 없이 이를 받아들이므로 이제 1.5라는 값을 가진다.

```
print(number.length);
```

이제 숫자의 .length 프로퍼티로 접근을 시도한다. 이 시점에서 number 변수의 타입은 숫자이고, 어떤 길이도 갖고 있지 않기 때문에 number.length 값은 [undefined]가 된다.

이제 JavaScript에는 알려지지 않은 값 개념이 있음을 확인할 수 있는데, 이는 [undefined]로 설명된다. 변수가 정의되지 않으면 그 안에 어떤 것으로도 접근할 수 없고, JavaScript는 변수를 오류로 생각하게 될 것이다.

```
number = number + " is a number? no, it is now a string";
print(number);
```

이제 number를 문자열과 결합하여 전체를 효과적으로 문자열로 만들었다. 이제 number.length가 인터프리터에 의해 정의되고 39의 값을 가지며 그 안에 39개의 문자가 있음을 보여준다.

```
number = (number.length == 0)
print(number);
```

여기서 우리는 표현식으로부터 온 Boolean 값을 number로 할당한다 (number.length==0). number.length는 0이 아니므로 표현식은 false를 리턴하고, 0으로 출력된다. 이것이 true라면 1로 출력될 것이다.

```
number = undefined
print(number);
```

이제 숫자를 undefined로 설정했는데, 이는 거꾸로된 단어(reserved word)이므로 앞에서 보인 바와 같이 설정할 수 있겠다.

재밌지 않은가?

팝 퀴즈 - 이제 값은 무엇인가?

Q1. 코드의 끝마다 모든 데이터 타입의 할당이 실행되고 난 후 number.length의 값은 무엇인가? 아래에서 선택하라 (문자열로 할당한 직후에는 39의 값을 가졌음을 기억하라).

1. 숫자를 undefined으로 설정했으므로 0이 값이다.
2. 숫자를 undefined으로 설정했으므로 undefined가 값이다.
3. 정의되지 않은 값으로부터 .length로 접근을 시도하므로 JavaScript는 오류라고 생각할 것이다.

반복 제어하기

프로그래밍 시 거의 대부분의 경우 특정 코드 부분을 반복적으로 실행해야 한다. 이는 코드 내에 반복 제어(루프 또는 반복 제어로 알려짐)를 포함시킴으로써 실행한다. JavaScript에서는 꽤 쉽다.

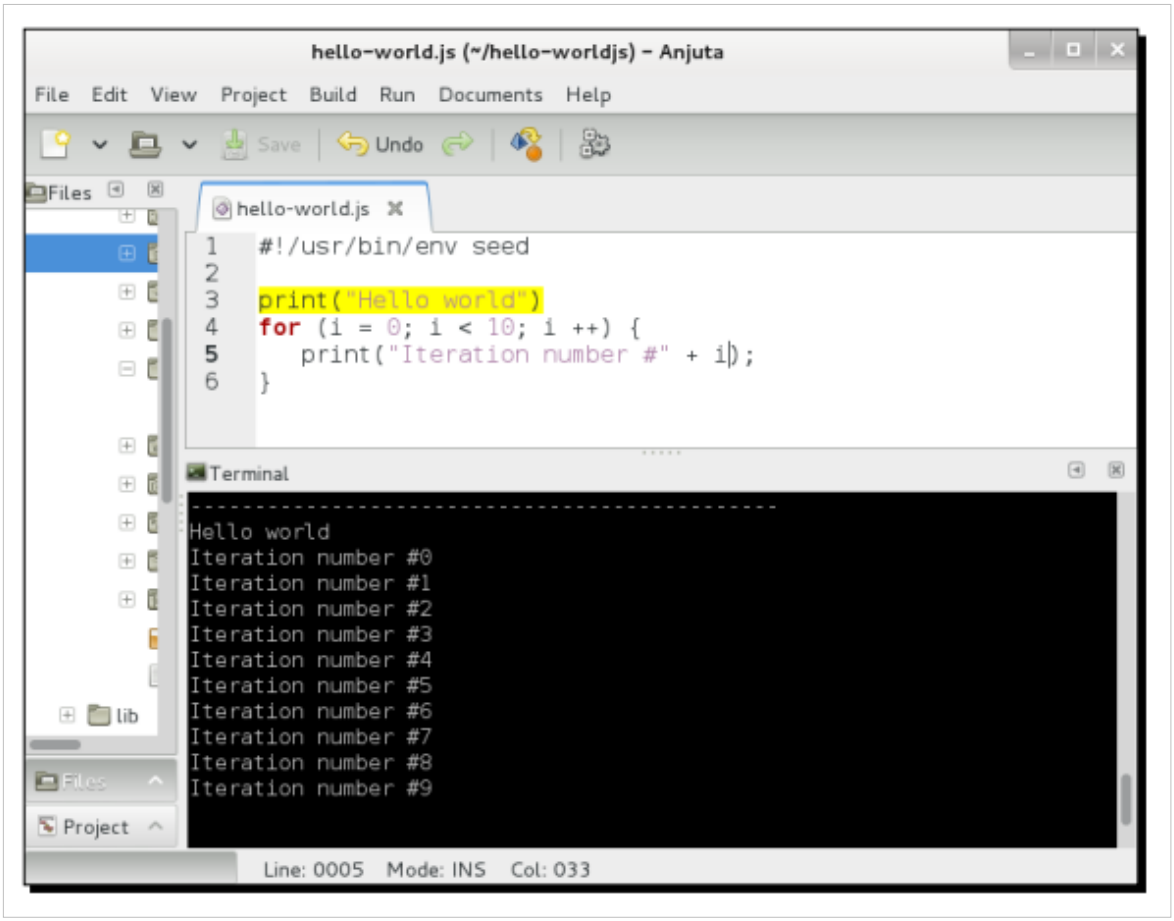
실행하기 - 반복 제어하기

반복을 제어하려면 아래 단계를 실행한다.

1. hello-world-iteration.js라는 새로운 파일을 생성하여 아래 코드를 넣어라.

```
#!/usr/bin/env seed

print("Hello, world")
for (i = 0; i < 10; i ++) {
    print("Iteration number #" + i);
}
```

2. 파일을 실행하라.
3. 텍스트가 10회 출력됨을 확인할 수 있다.



무슨 일이 일어났는가?

코드에서 우리는 JavaScript에게 for 루프를 이용해 10 회 반복할 것을 알린다. i의 값을 초기에 설정하기 때문에 (코드에서는 i=0) JavaScript는 1이 아니라 0부터 인덱스를 시작함을 알 수 있다. 각 반복에서 1을 i로 더한다 (for 루프에서 i++ 표현식을 보면 "i 값을 1씩 증가시켜라"는 의미다). i 값이 한계, 즉 10을 깨면(break the constraint) 루프는 즉시 멈춘다. 루프의 끝에는 i 값이 10이다. 10은 10보다 적은 수가 아니므로 (코드에서는 i<10를 넣었다) 루프를 깬다. 따라서 텍스트는 0부터 10까지가 아니라 0부터 9까지 표시한다.

시도해보기 - 내려세기

이제 올려세기가 끝났다. 그렇다면 내려세기는 어떨까?

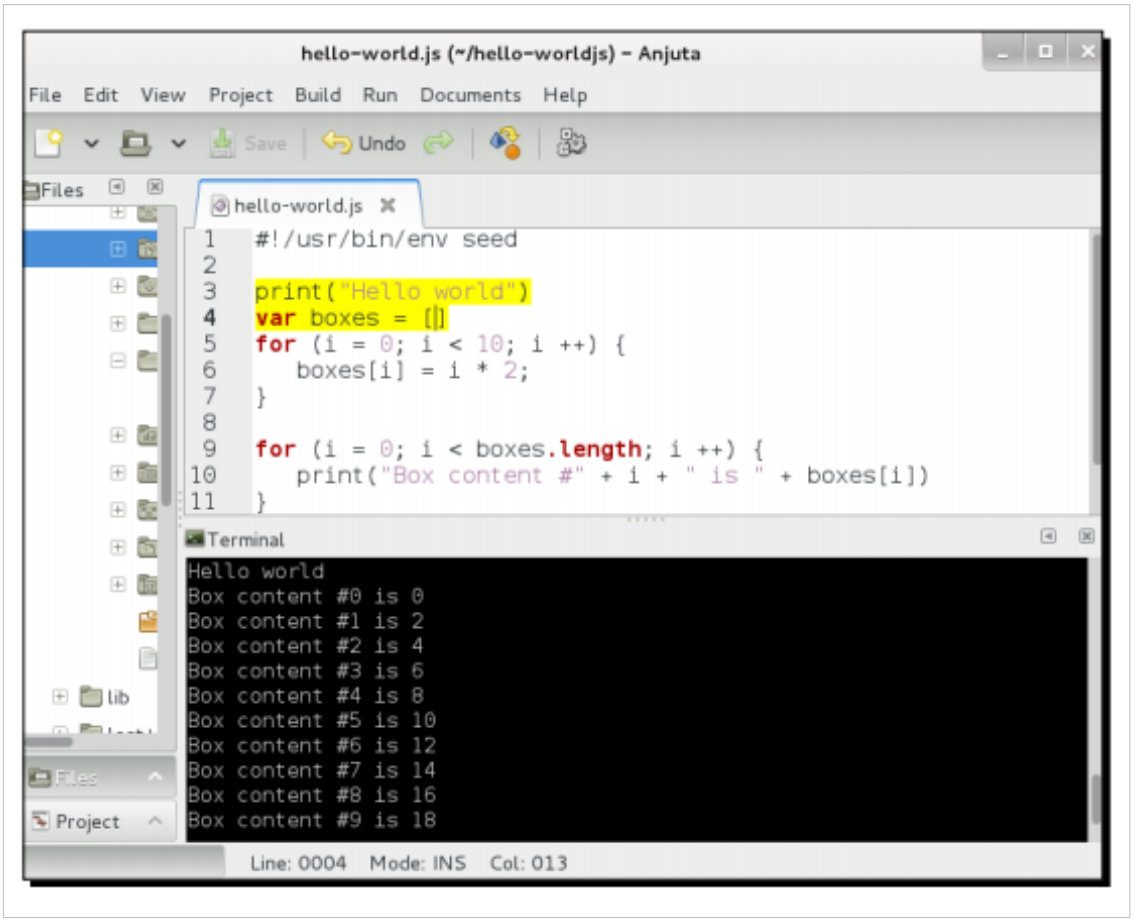
실행하기 - 배열 조작하기

배열은 동일한 타입으로 된 다수의 항목을 보유할 수 있는 박스의 집합체라고 가정해보자. 그리고 그러한 박스를 채워보자.

1. hello-world-array.js라는 새로운 스크립트를 생성하고 아래와 같이 채워라.

```
#!/usr/bin/env seed

print("Hello world")
var boxes = []
for (i = 0; i < 10; i ++) {
    boxes[i] = i * 2;
}

for (i = 0; i < boxes.length; i ++) {
    print("Box content #" + i + " is " + boxes[i])
}
```

2. 스크립트를 실행하라.
3. 박스 번호와 내용을 명시하는 텍스트가 보일 것이다.



무슨 일이 일어났는가?

가장 먼저 할 일은 boxes를 배열로 선언하는 일이다.

```
var boxes = []
```

배열의 크기는 설정하지 않았음을 기억하라. 그저 그것이 배열이라는 것만 밝히고 JavaScript가 알아서 하도록 내버려 두었을 뿐이다. 이는 우리가 배열의 내용을 수정할 때마다 언제든 배열이 줄어들거나 늘어날 수 있기 때문이다. 박스의 내용을 채우도록 한다.

```
for (i = 0; i < 10; i ++) {
    boxes[i] = i * 2;
}
```

그리고 각 박스에 값을 배열의 인덱스에 2를 곱한 값으로 설정한다. C 프로그래밍 언어에서와 같이 무언가를 할당하지 않고 인덱스 i에 내용을 직접 설정했다. 이후 배열의 내용을 출력하였다.

```
for (i = 0; i < boxes.length; i ++) {
    print("Box content #" + i + " is " + boxes[i])
}
```

배열의 길이는 객체 내의 length 변수로부터 얻을 수 있다. 이번 경우 boxes.length로부터 길이를 얻을 수 있다. 따라서 배열을 채울 때마다 길이가 자동으로 조정된다.

시도해보기 - 원하는 내용으로 채워보기

앞 절에서는 박스를 숫자로 채우는 작업을 시도했는데, 다른 데이터 유형은 어떨까? 박스를 문자열, 심지어 문자열과 정수를 섞어서 채워보도록 한다. 결과가 놀라운가?

JavaScript를 이용한 객체 지향 프로그래밍[OOP]

객체 지향 프로그래밍에 이미 익숙하다면 JavaScript에서 OOP는 무언가 제한적이고, 일반적인 OOP 실제를 따르지 않다는 사실에 준비되어야 한다. 이는 언어 자체가 전체적인 OOP 언어가 아니기 때문이다. 따라서 우리는 JavaScript 가 가진 한계 내에서 OOP 개념을 조정하고자 노력 중이다.

실행하기 - JavaScript 객체 이용하기

이제 가장 중요한 부분, JavaScript 객체를 맛볼 때가 되었다. 객체는 이번 책에서 광범위하게 사용할 예정이다. 먼저 간단한 객체를 하나 소개하겠다.

1. hello-world-object.js라는 새 스크립트를 생성하고 아래 코드로 채워라.

```
#!/usr/bin/env seed

print("Hello world")
var book = {};
print(book);
print(book.isbn);

book.isbn = "xxxx-1234-1234";
book.title = "A somewhat interesting book"
print(book);
print(book.isbn);
print(book.title);
```

2. 스크립트를 실행하라.
3. 출력된 값을 확인하라.

```
Hello world
[object Object]
[undefined]
[object Object]
xxxx-1234-1234
A somewhat interesting book
```

무슨 일이 일어났는가?

여느 데이터 유형과 마찬가지로 변수를 쉽게 객체로 할당할 수 있다.

```
var book = {};
```

이 행에서 우리는 book을 빈 객체로 정의한다. 여기서 객체는 중괄호로 초기화된다. 이 객체는 가장 단순한 형태에 속한다.

```
print(book);
```

이는 객체가 Object 타입임을 알리는 [object Object]를 출력한다.

```
print(book.isbn);
```

여기서 객체의 .isbn 프로퍼티로 접근을 시도하지만 비어 있기 때문에 처음엔 [undefined]가 된다.

```
book.isbn = "xxxx-1234-1234";
book.title = "A somewhat interesting book"
```

여기서 객체의 프로퍼티에 다른 값을 할당한다. 이 시점에서는 객체 안에 어떤 것이든 넣어도 좋으며, 객체를 사용하기 전에 초기화하거나 선언하지 않아도 된다.

```
print(book);
```

이때 다시 book 변수를 출력하는데, 여전히 Object 타입의 객체라고 말하고 있다.

```
print(book.isbn);
print(book.title);
```

하지만 값이 할당되고 출력이 가능해지므로 상황은 변한 셈이다.

권력엔 책임이 따른다

아마도 눈치를 이미 챘겠지만 JavaScript에서는 데이터 유형과 관련해 어떤 행위든 마음대로 할 수 있다. 하지만 이렇게 힘이 많을 때에는 코드의 정확도를 철저하게 검사해야 한다. JavaScript를 이용해 엉성하게 프로그래밍을 할 경우 코드가 증가하면서 오류를 추적하는 일이 더 힘들어지기 때문에 끔찍하게 변한다. JavaScript는 대·소문자에 민감하기 때문에 오류를 추적하기가 더 힘들어질 것이다. 코드 어딘가에 아래와 같은 행을 설정했다고 가정하자.

```
book.authorFirstName = "Random Joe"
```

그리고 코드의 다른 부분에서 이 행을 이용해 변수를 수정해보자.

```
book.authorFirstname = "Another Joe"
```

우리는 객체 안에 원하는 것을 마음대로 설정할 수 있기 때문에 JavaScript는 별다른 불평을 하지 않을 것이다. 하지만 앞에서와 같이 오식의 실수를 하지 않도록 코드를 두 번, 아니, 세 번 검사해야 할 책임이 있다. 그건 그렇고 지금 이야기 중인 버그는 발견했는가?

시도해보기 - 객체를 채우는 또 다른 방법

앞의 코드 일부를 아래와 같은 모습으로 수정하고 어떤 일이 일어나는지 살펴보라.

```
var book = {
    isbn:"xxxx-1234-1234",
    title:"A somewhat interesting book"
}
```

이제 우리는 다른 방식으로 ISBN과 제목을 객체 내에서 정의하고 있다. 여기서는 등호 대신 콜론을 이용하고, 정의부 사이에 콤마를 넣었다. 이러한 표기법을 JSON(JavaScript 객체 표기법)이라 부른다.

객체 구성하기

가장 단순한 형태의 JavaScript 객체를 사용했으니 좀 더 복잡한 객체를 사용해보자. 이는 JavaScript를 이용한 객체 지향 프로그래밍의 모험이 되겠다.

실행하기 - 생성자 갖고 놀기

객체의 생성을 이야기할 때는 생성자라는 특수 함수를 호출함을 의미한다. 그 방법을 살펴보자.

1. hello-world-constructor.js라는 새 파일을 생성하고 아래 코드로 채워라.

```
#!/usr/bin/env seed
print("Hello world")
var Book = function(isbn, title) {
   this.isbn = isbn;
    this.title = title;
}

book = new Book("1234", "A good title");
print(book.isbn);
print(book.title);
```

2. 파일을 실행하라.
3. 출력된 값을 확인하라.

```
Hello world
1234
A good title
```

무슨 일이 일어났는가?

앞의 코드와 사실상 비슷하지만, 먼저 클래스로 정의한 다음 객체로 인스턴스화한다는 점에서 차이가 있다.

```
var Book = function(isbn, title) {
    this.isbn = isbn;
    this.title = title;
}
```

이는 Book 클래스의 생성자다. 그 안에서 우리는 생성자 함수 내에 인자로 전달된 isbn 변수를 .isbn 프로퍼티에 할당한다. .title 프로퍼티에도 같은 일이 발생한다.

```
book = new Book("1234", "A good title");
```

여기서 제공된 인자로 Book 클래스를 인스턴스화함으로써 book(모두 소문자다!)이라는 새로운 변수를 생성한다.

```
print(book.isbn);
print(book.title);
```

이제 .isbn과 .title 의 값이 출력되는 것을 볼 수 있다.

클래스와 객체

클래스는 앞에 소개한 코드에 포함된 var Book = function(..) {...} 처럼 하나의 정의에 불과하며, 후에 new 연산자를 이용해 실제로 인스턴스화하기 전에는 객체가 아니다. 클래스가 객체가 되면 우리는 이것을 클래스의 인스턴스라고 부른다. 앞서 우리는 클래스 정의 없이 중괄호만 이용해 다른 방식으로 인스턴스화를 실행하였다.

관례상 주로 대문자와 소문자가 섞인 CamelCase를 이용해 클래스를 명명하는데, 첫 번째 단어의 첫 문자는 대문자로 시작한다(예: Book). 반대로 객체 인스턴스나 변수의 경우 첫 단어의 첫 문자에 소문자를 이용한다(예: book).

팝 퀴즈 - 차이점이 눈에 띄는가?

아래 코드를 살펴보자.

```
var Circle = function(radiusInPixel) {
    this.radius = radiusInPixel
}

var circle = new Circle(100);
```

Q1. circle은 무엇이고 Circle은 무엇인가? 아래에서 무엇이 올바른 문(statement)인가?

1. Circle은 정의를 갖고 있기 때문에 클래스이고, circle은 Circle 클래스로부터 인스턴스화된 객체다.
2. circle은 정의를 갖고 있기 때문에 객체이고, Circle은 circle 객체의 인스턴스다.

프로토타입 이용하기

OOP에서는 객체에 부착된 함수나 메서드를 가질 수 있다. 즉, 함수가 메모리에서 특정 객체에 한정된다는 말이다. 하나의 객체에서 어떤 함수를 호출하면 동일한 함수를 가진 동일한 타입의 다른 객체를 간섭하지 않는다는 의미다. JavaScript에서는 이러한 기능을 획득하기 위해 프로토타입을 이용한다.

실행하기 - 프로토타입 추가하기

이제 몇 가지 메서드를 클래스로 추가해보자. 여기서는 prototype 객체를 이용해 메서드를 정의해보겠다.

1. hello-world-prototype.js라고 불리는 새로운 스크립트를 생성하고 아래 내용으로 채워라.

```
#!/usr/bin/env seed

print("Hello world")

var Book = function(isbn, title) {
    this.isbn = isbn;
```

```
    this.title = title;
}

Book.prototype = {
    printTitle: function(){
        print("Title is " + this.title);
    },

    printISBN: function() {
        print("ISBN is " + this.isbn);
    }
}

var book = new Book("1234", "A good title");
book.printTitle();
book.printISBN();
```

2. 스크립트를 실행하라.
3. 출력된 값을 확인하라.

```
Hello world
Title is A good title
ISBN is 1234
```

무슨 일이 일어났는가?

JavaScript 객체에서 prototype은 클래스나 객체 내의 모든 프로퍼티와 메서드를 보유하는 특수 객체다. 따라서 지금은 프로토타입을 우리만의 메서드로 채우고자 한다.

```
var Book = function(isbn, title) {
    this.isbn = isbn;
    this.title = title;
}
```

이 코드에는 앞에서와 동일한 생성자가 있다.

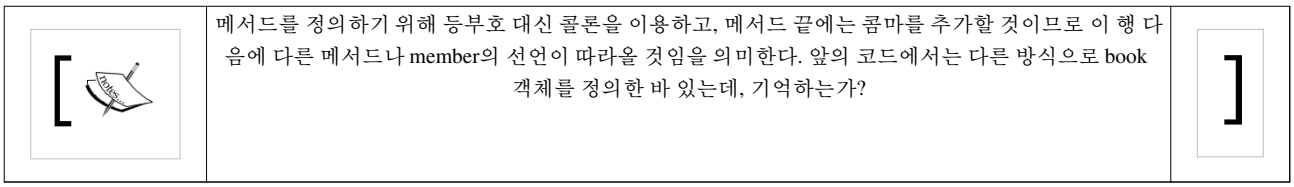
```
Book.prototype = {
```

이후 프로토타입의 선언을 시작하고, 고유의 메서드 선언으로 채울 준비를 한다.

```
    printTitle: function(){
        print("Title is " + this.title);
    },
```

여기서 함수 본체에서 설명하듯이 첫 번째 메서드를 넣는다.

```
    printISBN: function() {
        print("ISBN is " + this.isbn);
    }
```



이제 다음 메서드가 따라온다. 여기서는 콤마를 넣지 않고 정의를 끝마친다.

```
var book = new Book("1234", "A good title");
```

그 다음에 명시된 인자와 함께 Book 객체를 생성함으로써 book 변수를 선언한다.

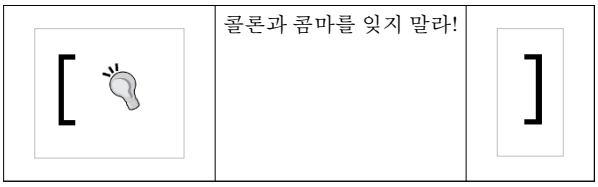
```
book.printTitle();
book.printISBN();
```

마지막으로 메서드를 호출하여 사용한다(메서드명 다음에 괄호를 주목하라).

시도해보기 - 더 많은 메서드 추가하기

메서드를 좀 더 추가해보는 건 어떨까? 가령 다음과 같은 메서드가 필요하다고 가정하자.

- isbn을 리턴하는 getISBN()
- 책 제목을 리턴하는 getTitle()



실행하기 - 객체의 프로토타입 수정하기

앞서 언급했듯이 클래스가 아니라 객체의 프로토타입에 직접 무언가를 넣을 수도 있다. 이것은 일상생활에서 흔히 하는 일은 아니지만 알아두면 후에 용이하게 작용할 것이기에 학습하고자 한다. 우리 프로토타입에 정의된 함수를 런타임 시 다른 함수로 대체하길 원한다고 가정해보자!

1. hello-world-proto.js라는 새로운 스크립트를 생성하고 아래 내용으로 채워라.

```
#!/usr/bin/env seed

print("Hello world")

var Book = function(isbn, title) {
    this.isbn = isbn;
    this.title = title;
}

Book.prototype = {
    printTitle: function(){
        print("Title is " + this.title);
    },

    printISBN: function() {
        print("ISBN is " + this.isbn);
    }
```

```
}

var book = new Book("1234", "A good title");

book.printTitle();
book.printISBN();

book.__proto__ = {
    author: "Joe Random",
    printAuthor: function() {
        print("Author is " + this.author);
    }
}

book.printAuthor();

var anotherBook = new Book("4567", "A more better title");

anotherBook.printTitle();
anotherBook.printISBN();
anotherBook.printAuthor(); // this is invalid
```

2. 스크립트를 실행하라.
3. 첫 번째 책의 저자를 출력하지만 두 번째 책의 출력은 실패함을 주목하라.

```
Hello world
Title is A good title
ISBN is 1234
Author is Joe Random
Title is A more better title
ISBN is 4567
** (seed:4911): CRITICAL **: Line 39 in hello-world.js:
TypeError 'undefined' is not a function (evaluating 'anotherBook.
printAuthor()')
```

무슨 일이 일어났는가?

런타임 시 프로토타입을 수정하기 위해서는 알아야 할 작은 비밀이 하나 있다. 프로토타입은 더 이상 prototype 프로퍼티가 아니라 __proto__를 이용해야 접근할 수 있다는 점이다. 아래 행에서는 book 객체를 인스턴스화한다.

```
var book = new Book("1234", "A good title");
```

그리고 __proto__를 이용해 접근하는 프로토타입 안에 두 개의 프로퍼티를 추가한다.

```
book.__proto__ = {
    author: "Joe Random",
    printAuthor: function() {
        print("Author is " + this.author);
    }
```

```
}
```

바로 사용해보자.

```
book.printAuthor();
```

하지만 다른 인스턴스에서는 이를 실행할 수 없었다. 이유를 알겠는가? 그렇다, 우리는 book 객체만 수정했기 때문에 anotherBook 객체엔 영향을 미치지 않기 때문이다.

```
var anotherBook = new Book("4567", "A more better title");

anotherBook.printAuthor(); // this is invalid
```

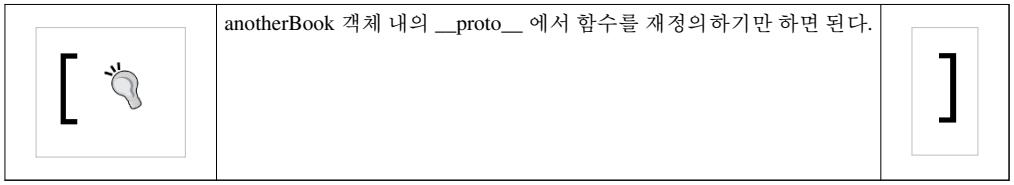
팝 퀴즈 - 전역적으로 만드는 방법은?

Q1. Book 클래스로부터 생성된 모든 객체에 printAuthor 메서드를 추가하는 최선의 방법은 무엇인가?

1. 생성된 모든 객체에서 __proto__에 printAuthor를 추가하면 모든 객체에서 이용 가능한 함수를 갖게 될 것이다.
2. Book 클래스 프로토타입에서 printAuthor를 추가하면 Book으로부터 생성된 모든 객체는 함수를 갖게 될 것이다.

시도해보기 - 구현부의 세부 내용 변경하기

anotherBook 객체는 특별한 책을 선언하는 데에만 사용하길 원한다고 가정해보자. 너무 특별하여서 printTitle 함수에 <book-title>을 책 제목으로 하고 <book-title> is a really good title 이라는 내용을 출력하길 원한다고 치자.



모듈화

커다란 프로젝트를 구현하여 단일 스크립트 안에 넣었다고 상상해보자. 디버깅이 엄청나게 까다로워지기 때문에 머지않아 끔찍한 일이 될 것이다. 따라서 코드가 더 커지기 전에 논해보도록 하자.

실행하기 - 프로그램 모듈화하기

이제 우리 소프트웨어를 모듈화할 것이다.

1. hello-world-module.js 라는 새로운 파일을 생성하여 아래 내용으로 채우자.

```
#!/usr/bin/env seed

print("Hello world")

var BookModule = imports.book

var book = new BookModule.Book("1234", "A good title");
book.printTitle();
book.printISBN();
```

2. book.js이라는 또 다른 새 스크립트를 생성하고 아래 내용으로 채워라.

```
var Book = function(isbn, title) {
    this.isbn = isbn;
    this.title = title;
}

Book.prototype = {
    printTitle: function(){
        print("Title is " + this.title);
    },

    printISBN: function() {
        print("ISBN is " + this.isbn);
    }
}
```

3. (book.js 가 아니라) hello-world-module.js 를 실행하라.
4. 출력 내용을 확인하라.

무슨 일이 일어났는가**?**

출력된 결과를 살펴보면 앞의 코드와 정확히 똑같음을 볼 수 있다. 하지만 여기서는 코드를 두 개의 파일로 나누겠다.

```
var BookModule = imports.book

var book = new BookModule.Book("1234", "A good title");
```

여기서 imports 명령을 이용한 book의 결과와 함께 BookModule 변수에 부착할 것을 Seed로 요청한다. 이 때는 현재 우리 디렉터리에 book.js가 있을 것으로 기대한다. 그래야만 book.js 내 모든 객체들을 BookModule 변수로부터 접근할 수 있다. 따라서 우리는 앞의 행을 이용해 book 객체를 구성한다.

뿐만 아니라 book.js에 더 이상 해쉬뱅(hashbang) 행이 없음을 주목하라. book.js 가 아니라 hello-world-module.js 를 진입점으로 사용하기 때문에 해쉬뱅은 필요하지 않은 것이다.

이 접근법을 이용하면 파일에 객체를 배치시켜 필요할 때마다 가져올 수 있다. 메모리 사용이 효율적으로 변할 뿐 아니라 코드 구조체를 깨끗하게 유지하기도 한다.

이로써 GNOME 애플리케이션 개발 프로그래밍 언어로서 JavaScript에 관한 간략한 소개는 끝이 난다. 이제 Vala로 넘어가 보자.

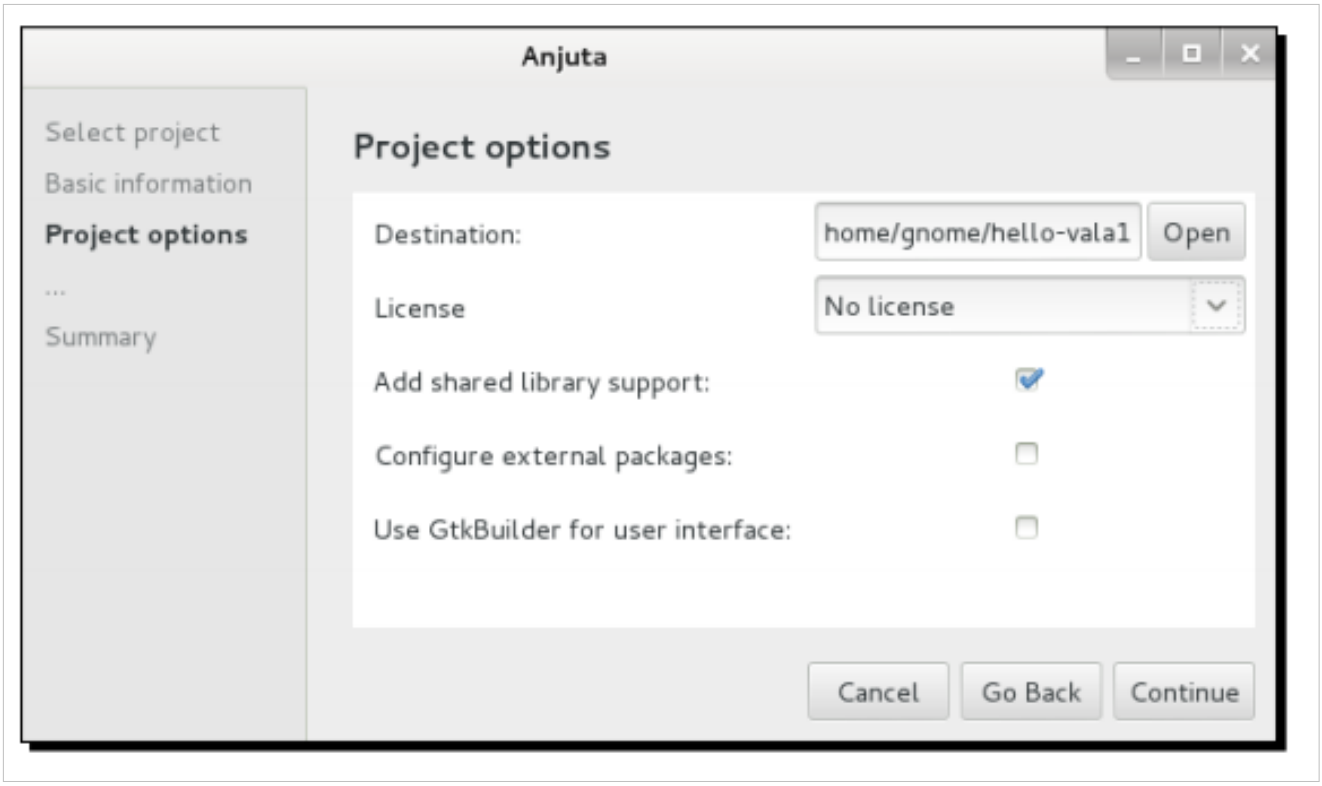
Vala 알아가기

Vala는 JavaScript에 비해 새로 생긴 언어로, 그러한 개념이 생긴 이후부터 GNOME 개발에서 유일하게 사용하는 언어에 해당한다. 이것은 꽤 흥미로운 개념을 바탕으로 하는데, 바로 프로그래머들이 C#와 Java와 닮은 구문에 노출되지만 그 바닥에서는 코드가 순수 C 언어로 번역되어 바이너리로 컴파일될 것이다.

이 덕분에 GNOME 프로그래밍에 좀 더 쉽게 접근할 수 있는데, C를 이용한 GNOME 애플리케이션의 개발은 초보자들이 이해하기엔 꽤 까다롭기 때문이다. 수많은 표준(boilerplate) 코드 조각을 복사하여 자신의 소스 코드로 복사한 다음 지침서에 따라 수정해야 하는 작업이 요구되기 때문이다. Vala에선 이러한 단계가 완전히 숨겨진다.

JavaScript에서 시도했던 모험과 비슷하게 어떠한 그래픽 요소도 구현하지 않고 Vala 언어의 기본을 학습하고자 한다. Vala는 모든 특성을 갖춘(full-blown) 프로그래밍 언어로, Vala를 학습하는 내내 OOP 개념을 사용할 것이다.

이제 실험으로 사용하게 될 프로젝트를 준비해보자. 제 2장, 무기 준비하기에서 소개한 단계를 기억하는가? 좋다! 약간 내용을 변경하여 단계를 실행해보자. hello-vala를 프로젝트명으로 사용하겠다.



위의 스크린샷을 보면 Project options에서 No license를 선택해 다음으로 실행할 수정내용을 최소화하였다. Vala의 핵심을 이해하도록 간단한 텍스트 기반의 애플리케이션을 실행하길 원하므로 Use GtkBuilder for user interface 옵션을 체크 해제한다.

실행하기 - 프로그램의 진입점

애플리케이션을 처음부터 만드는 것이 무엇인지 이해할 수 있도록 생성된 코드를 모두 우리만의 코드로 대체할 것이다.

1. 생성된 hello_vala.vala 파일을 편집하고 아래 내용으로 채워라.

```
using GLib;

public class Main : Object
{
    public Main ()
    {
    }

    static int main (string[] args)
    {
        stdout.printf ("Hello, world\n");
        return 0;
    }
}
```

2. Run 메뉴를 클릭하고 Execute를 선택하라.

3. 출력된 텍스트를 확인하라.

```
Hello, world
```

무슨 일이 일어났는가**?**

Book 클래스를 살펴보는 것으로 시작한다.

```
using GLib;
```

이 행은 우리가 GLib 네임스페이스를 사용 중이라고 말한다.

```
public class Main : Object
```

이는 Main 클래스의 정의다. GLib.Object 클래스로부터 파생되었음을 표시한다. 첫 행에서 이미 GLib 네임스페이스를 사용하고 있음을 명시했기 때문에 GLib.Object라는 풀네임이 아니라 Object만 쓴다.

```
public Main ()
{
}
```

앞의 구조체는 클래스의 생성자다. 이제 빈 구조체가 생겼다.

```
    static int main (string[] args)
    {
        stdout.printf ("Hello, world\n");
        return 0;
    }
}
```

이것이 우리 프로그램의 진입점이다. 정적으로 선언되면 main 함수는 애플리케이션에서 실행될 첫 번째 함수로 간주될 것이다. 이 함수가 없이는 애플리케이션을 실행할 수가 없다.

또 한 가지, 정적인 main 함수는 하나만 존재해야 하는데, 그렇지 않으면 프로그램이 컴파일하지 않을 것이기 때문이다.

시도해보기 - 생성된 C 코드 살펴보기

이제 생성된 C 코드를 src/ 디렉터리에서 이용할 수 있을 것이다. Files dock를 이용해 파일시스템을 탐색하여 hello_vala.c를 찾아라. 이를 열고 Vala가 어떻게 Vala 코드를 C 코드로 변형하는지 확인하라.

C 코드를 수정할 수도 있지만 Vala 코드를 변경할 때마다 변경된 내용을 덮어쓸 것이며, C 코드가 재생성될 것이다.

Member 접근 명시자

Vala는 member 접근 명시자 집합을 정의하는데, 이를 이용하면 다른 클래스 또는 그로부터 상속된 클래스로부터 member로 접근할 수 있을 것인지 정의할 수 있다. 이러한 용어는 사용하기 쉽고 깔끔한 애플리케이션 프로그래밍 인터페이스(API) 집합을 만들 수 있는 방법을 제공한다.

실행하기 - member 접근성 정의하기

클래스 member로의 접근성을 어떻게 명시하는지 살펴보자.

1. 새로운 파일을 생성하고 src/ 디렉터리에 book.vala로 저장하라. 파일을 아래 내용으로 채워라.

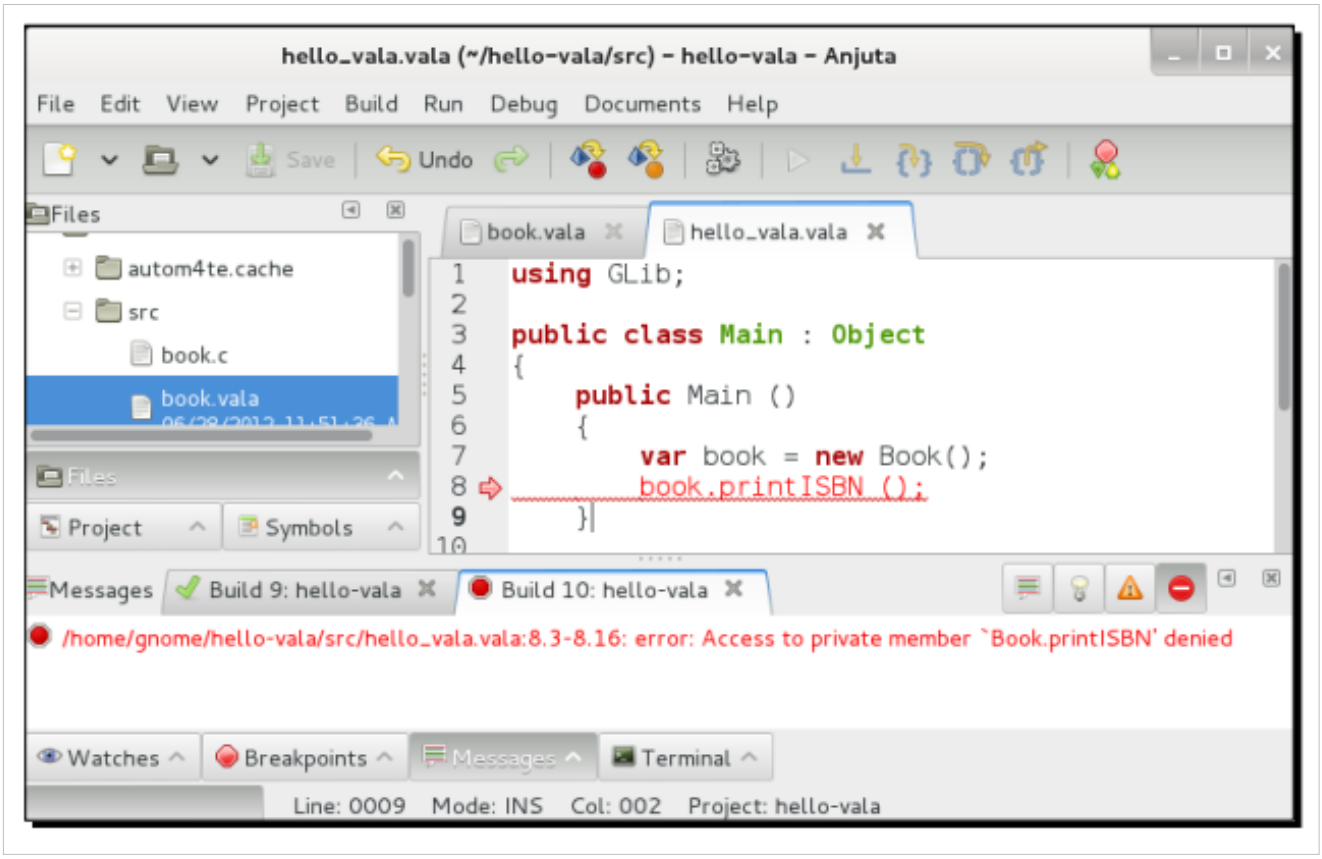
```
using GLib;

public class Book : Object {
    private string title;
        private string isbn;

    public Book(string isbn, string title) {
            this.isbn = isbn;
            this.title = title;
    }

    public void printISBN() {
    stdout.printf("%s\n", isbn);
    }

    public void printTitle() {
        stdout.printf("%s\n", title);
    }
}
```

2. 이를 프로젝트로 추가할 필요가 있다. Project 메뉴를 클릭하고 Add Source File...을 선택하라.
3. 다음 대화상자에서 Target 옵션을 클릭하고 src/ 안에서 hello_vala를 찾은 다음 그 아래의 파일 선택 박스에서 book.vala를 선택하라.
4. hello_vala.vala의 main 함수가 아래와 같은 모습이 되도록 수정하라.

```
using GLib;

public class Main : Object
{
    public Main ()
    {
        var book = new Book("1234", "A new book");
        book.printISBN ();
    }

    static int main (string[] args)
    {
        stdout.printf ("Hello, world\n");
        var main = new Main();
```

```
        return 0;
    }
}
```

5. 파일을 실행하라.
6. 프로그램을 빌드할 수 없음을 주목하라.



무슨 일이 일어났는가**?**

오류 메시지를 확인하면 Book.printISBN로 호출하려는 접근을 모두 거부함을 볼 수 있다(이러한 점 표기법은 Book 클래스로부터 printISBN member로 읽는다).

```
var book = new Book("1234", "A new book");
book.printISBN ();
```

Main 클래스 생성자의 내용은 다음과 같다. 여기서 우리는 Book을 book 변수로 인스턴스화하고 printISBN을 호출한다.

```
void printISBN() {
stdout.printf(isbn);
}
```

하지만 Book 클래스의 내용은 위와 같다. 별 문제가 없어 보이지만 클래스 함수로부터 이 함수로 접근할 수 없도록 만드는 데 중요한 내용이 빠진 것으로 밝혀졌다.

접근 명시자

Vala가 인식하는 접근 명시자의 리스트는 다음과 같다.

- private: 클래스 또는 구조체 내로 접근이 제한된다.
- public: 접근이 제한되지 않는다.
- protected: 클래스 내로 그리고 그 클래스로부터 상속된 클래스 내로 접근이 제한된다.
- internal: 패키지 내부의 클래스 내로 접근이 제한된다.

어떤 것도 명시하지 않으면 접근은 기본적으로 private으로 설정된다. 이 때문에 프로그램을 빌드할 수 없는 것이다.

팝 퀴즈 - 이를 어떻게 고칠까?

앞서 언급하였듯 printISBN 함수 앞에 어떤 명시자도 넣지 않기 때문에 private한 것으로 간주된다. 이러한 상황은 printISBN 함수 안에 올바른 접근 명시자를 넣음으로써 고칠 수 있다.

Q1. 아래 옵션 중 어떤 명시자가 올바르다고 생각하는가?

1. public. Book 클래스 밖에 있는 Main 클래스로부터 접근하길 원하기 때문이다.
2. 없다. 그저 Main 생성자에서 printISBN을 호출하는 방법을 수정하길 원할 뿐이다.

기본 데이터 유형

이제 나아가 이용 가능한 기본 데이터 유형을 학습할 차례이므로 문자열, 숫자, Boolean과 상호작용하는 방법을 살펴볼 것이다.

1. bookstore.vala라는 새로운 파일을 생성하고 src/ 에 넣어라. 파일을 아래 내용으로 채워라.

```
using GLib;

public class BookStore {
    private Book book;
    private double price = 0.0;
    private int stock = 0;
    public BookStore (Book book, double price, int stock) {
        this.book = book;
        this.price = price;
        this.stock = stock;
    }

    public int getStock() {
        return stock;
    }

    public void removeStock(int amount) {
        stock = stock - amount;
    }

    public void addStock(int amount) {
        stock = stock + amount;
    }

    public double getPrice() {
        return price;
```

```
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public bool isAvailable() {
        return (stock > 0);
    }
}
```

2. 이 파일을 프로젝트로 추가하라.
3. 우리 Main 클래스를 아래와 같은 모습이 되도록 수정하라.

```
using GLib;

public class Main : Object
{
    public Main ()
    {
        var book = new Book("1234", "A new book");
        book.printISBN ();

        var store = new BookStore(book, 4.2, 10);
        stdout.printf ("Initial stock is %d\n", store.getStock());
        stdout.printf ("Initial price is $ %f\n", store.getPrice());
        store.removeStock(4);
        store.setPrice(5.0);
        stdout.printf ("Stock is %d\n", store.getStock());
        stdout.printf ("and price is now $ %f\n", store.getPrice());
        var status = "still available";
        if (store.isAvailable() == false) {
            status = "not available";
        }
        stdout.printf ("And the book is %s\n", status);
    }

    static int main (string[] args)
    {
        stdout.printf ("Hello, world\n");
        var main = new Main();
        return 0;
    }
}
```

4. 파일을 실행하라.
5. 데이터가 어떻게 조작되고 출력되는지 확인하라.

```
Hello, world
1234
Initial stock is 10
Initial price is $ 4.200000
Stock is 6
and price is now $ 5.000000
And the book is still available
```

무슨 일이 일어났는가**?**

호출하는 코드, Main 생성자부터 분석을 시작해보자.

```
var store = new BookStore(book, 4.2, 10);
```

BookStore 클래스로부터 새로운 store 객체를 인스턴스화한다. book 객체, 부동 소수점수, 정수로 객체를 초기화한다.

```
public BookStore (Book book, double price, int stock) {
```

앞에서와 같이 BookStore 생성자에서 인자 리스트에 데이터 유형을 명시해야 한다. 이후 Book 객체, double 정밀도의 숫자, 정수를 수락하길 원한다고 말해야 한다.

```
this.book = book;
this.price = price;
this.stock = stock;
```

다음으로 book, price, stock이라는 private member들에게 인자를 할당한다. 여기서 인자로부터 book member를 private member로부터 book member로 할당하고 싶다는 것을 표기하기 위해 this. 를 이용한다. 인자 변수를 다른 이름, 가령 bookObject로 명명한다면 this를 생략해도 좋은데, 더 이상 이름이 모호하지 않아서 bookObject가 우리의 member가 아니라 인자 리스트로부터 비롯됨을 알기 때문이다. price와 stock에도 똑같은 일이 일어난다.

```
stdout.printf ("Initial stock is %d\n", store.getStock());
```

이것이 바로 printf 를 이용해 정수를 출력하는 방식이다. %d는 정수에 대한 플레이스홀더로 사용한다.

```
stdout.printf ("Initial price is $ %f\n", store.getPrice());
```

그리고 이것은 printf를 이용해 실수를 출력하는 방식이다. 실수에 대한 플레이스홀더로 %f를 사용한다.

```
store.removeStock(4);
```

이후 스톡에서 4개의 책을 제거한다. 내부적으로 이는 아래와 같이 BookStore.removeStock에서 정의된다.

```
stock = stock - amount;
```

단순히 정수라는 이유로 수학 표현식을 이용해 뺄셈을 실행한다.

```
var status = "still available";
if (store.isAvailable() == false) {
    status = "not available";
}
```

다음으로 Boolean 표현식 평가가 있다. 값이 false라면 status의 값을 변경한다. status에 대한 타입이 string일 경우 값을 그저 할당하면 된다.

```
stdout.printf ("And the book is %s\n", status);
```

마지막으로 우리 문자열 값을 printf에 넣기 위해 %s를 플레이스홀더로 사용한다.

Gee, 이게 뭘까**?**

Gee는 Vala로 쓴 컬렉션 라이브러리다. 컬렉션의 기본 타입으로는 리스트, 세트, 맵이 있다. 이들은 배열과 비슷하지만 강력한 기능들을 더 많이 갖고 있다.

실행하기 **- Gee** 라이브러리 추가하기

Gee를 좀 더 가까이 살펴보도록 하자. 하지만 먼저 이것을 프로젝트에 추가해보자.

1. Project 메뉴를 클릭하고 Add Library...를 선택하라.
2. Select the target for the library의 src/ 에서 hello_vala를 찾아라.
3. New library... 버튼을 클릭하라.
4. 리스트에서 gee를 찾아 대화상자 하단에 Module 옵션에서 체크한 다음 HELLO_VALA를 찾아라. 이는 Gee를 C 컴파일 단계로 추가함을 의미한다. 기본적으로 이 단계는 Gee를 빌드 시스템으로 추가하도록 configure.ac 파일을 수정하는 것이다.
5. 이후 Files dock의 /src 디렉터리에서 Makefile.am 을 찾아 열어라. hello_vala_VALAFLAGS stanza를 찾아 아래와 같은 내용으로 수정하라.

```
hello_vala_VALAFLAGS = \
--pkg gtk+-3.0 \
--pkg gee-1.0
```

6. Makefile.am 파일을 저장하고 닫아라. 이는 Gee를 Vala 컴파일 단계로 추가한다는 의미다.
7. Build 를 클릭하고 Clean Project를 선택하라. 빌드 시스템이 준비한 모든 스크립트와 생성된 코드로부터 프로젝트를 삭제하여(clean) Makefile.am 과 configure.ac 에서 변경한 내용을 모두 정리할 수 있을 것이다.
8. 앞의 코드를 다시 실행해보라. 더 이상 오류가 발생하지 않을 것이다.

무슨 일이 일어났는가**?**

Gee를 프로젝트로 추가하였다. Vala에 대한 Anjuta의 지원은 아직 완벽하지 않으므로, 라이브러리를 프로젝트로 추가하기 위해서는 C 컴파일용과 Vala 컴파일용으로 두 가지 액션(방금 보인 바와 같이)을 취해야 한다. 두 단계를 거치지 않으면 Vala가 Gee 네임스페이스를 인식할 수 없을 뿐 아니라 C 컴파일러가 Gee 헤더와 라이브러리를 찾을 수 없을 것이기 때문에 프로그램을 빌드할 수가 없다.

실행하기 **- Gee** 실행하기

프로젝트에 Gee를 실행하고 나면 Gee가 제공하는 기능을 빠르게 확인해보자. 그 중에서 간단한 배열 리스트부터 시작해보자.

1. book.vala가 아래와 같은 모습이 되도록 수정하라.

```
using GLib;
using Gee;

public class Book : Object {
        private string title;
        private string isbn;
    private ArrayList<string> authors;

    public Book(string isbn, string title) {
        this.isbn = isbn;
        this.title = title;
        authors = new ArrayList<string>();
    }

    public void addAuthor(string author) {
        authors.add(author);
    }

        public void printISBN() {
        stdout.printf("%s\n", isbn);
        }

    public void printTitle() {
        stdout.printf("%s\n", title);
    }

    public void printAuthors() {
        foreach (var author in authors) {
            stdout.printf("Author name: %s\n", author);
        }
    }
}
```

2. Main 클래스 생성자가 아래의 내용을 포함하도록 수정하라.

```
var book = new Book("1234", "A new book");
book.printISBN ();
book.addAuthor("Joe Random");
book.addAuthor("Joe Random Jr.");
book.printAuthors();
```

3. 위를 실행하라.
4. 책의 모든 저자를 출력하는지 확인하라.

```
Hello, world
1234
Author name: Joe Random
Author name: Joe Random Jr.
```

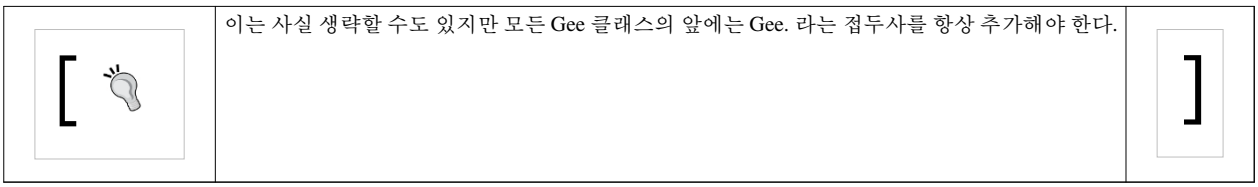
```
Initial stock is 10
Initial price is $ 4.200000
Stock is 6
and price is now $ 5.000000
And the book is still available
```

무슨 일이 일어났는가**?**

Gee가 제공하는 많은 컬렉션 데이터 구조체들 중 하나인 배열 리스트를 활용하고자 한다.

```
using Gee;
```

Gee를 사용하기 위해서는 먼저 우리가 Gee 네임스페이스를 사용하고 있음을 선언한다.



이제 Book 클래스에서 member 선언을 살펴보자.

```
public class Book : Object {
        private string title;
        private string isbn;
    private ArrayList<string> authors;
```

꺾쇠 괄호로 된 구조체는 일반적(generics) 프로그래밍이라 부른다. 이는 데이터 구조체에 포함된 데이터(지금 맥락에서는 ArrayList)는 일반적이란 의미다. 정수 타입의 배열이 있다면 ArrayList<int> 라는 식으로 넣을 것이다. 따라서 이러한 특정 행에 우리는 string 타입의 내용을 가진 ArrayList를 갖고 있으며, authors라는 이름으로 된 리스트를 호출한다. 생성자 내에서 우리는 아래의 구문을 이용해 배열 리스트를 초기화해야 한다.

```
public Book(string isbn, string title) {
    this.isbn = isbn;
    this.title = title;
    authors = new ArrayList<string>();
}
```

이는 string 타입으로 된 내용을 가진 ArrayList 객체를 할당할 필요가 있다는 의미다. 선언만으로는 충분하지 않음을 주목한다. 이 사실을 잊어버리면 프로그램은 충돌할 것이다.

```
public void addAuthor(string author) {
    authors.add(author);
}
```

여기서 우리는 ArrayList 클래스가 제공하는 add 함수를 이용한다. 이름에서 알 수 있듯이 데이터를 배열 리스트로 추가할 것인데, 문자열 내용으로 선언하고 초기화하기 때문에 string만 수락할 수 있음을 주목한다.

```
    public void printAuthors() {
        foreach (var author in authors) {
            stdout.printf("Author name: %s\n", author);
```

```
        }
    }
}
```

이제 배열 리스트의 내용을 반복한다. 각 반복마다 얻은 값을 author 변수로 할당하는 동안 반복하도록 foreach 명령을 이용한다. var author in authors 표현식을 이용함을 주목하라. author 변수를 string으로 명시하지는 않지만 대신 var 키워드를 이용해 자동 변수 생성을 이용한다. 이 행에서 authors 변수의 내용에 따라 var에 타입이 할당될 것이다. authors 내용 타입은 string이기 때문에 var 키워드에 바인딩된 author 변수는 string이 될 것이다. 이러한 유형의 생성은 컬렉션 또는 데이터 구조체에 저장된 데이터 유형에 따라 클래스가 어떤 데이터 유형이든 처리할 수 있도록 일반화할 때 매우 유용하다.

선언 중에 **member** 초기화하기

앞의 코드에서 생성자 내에 배열 리스트를 초기화한다. 아니면 생성자 내에서 초기화하지 않고 선언 영역에서 선언하는 동안 초기화하는 대안적인 방법이 있다. 이는 아래와 같이 실행하면 된다.

```
private ArrayList<string> authors = new ArrayList<string>();
```

코드가 증가하고 하나 이상의 생성자를 갖게 되면 모든 초기화 코드를 모든 생성자로 복사해야 하기 때문에 생성자 내에서 초기화하는 방법보다 위의 대안책이 더 낫다.

실행하기 - 시그널 기다리기

Vala는 시그널을 발생시키고 기다리기 위한 구조체를 갖고 있는데, 이는 코드에 무언가가 발생하면 정보를 구독(subscribe to)하는 메커니즘에 해당한다. 우리는 시그널로 어떤 액션을 실행하는 함수를 연결함으로써 시그널을 구독할 수 있다. 어떻게 작동하는지 살펴보자.

1. bookstore.vala 파일을 수정하고 두 개의 선언을 추가하라.

```
public class BookStore {
    ...
    public signal void stockAlert();
    public signal void priceAlert();
```

2. bookstore.vala의 removeStock과 setPrice 함수가 다음과 같은 모습이 되도록 수정하라.

```
public void removeStock(int amount) {
    stock = stock - amount;
    if (stock < 5) {
        stockAlert();
    }
}
public void setPrice(double price) {
    this.price = price;
    if (price < 1) {
        priceAlert();
    }
}
```

3. Main 구조체가 아래와 같은 모습이 되도록 수정하라.

```
public Main ()
    {
```

```
        var book = new Book("1234", "A new book");
        book.printISBN ();
        book.addAuthor("Joe Random");
        book.addAuthor("Joe Random Jr.");
        book.printAuthors();

        var store = new BookStore(book, 4.2, 10);

        store.stockAlert.connect(() => {
            stdout.printf ("Uh oh, we are going to run out stock
soon!\n");
        });

        store.priceAlert.connect(() => {
            stdout.printf ("Uh oh, price is too low\n");
        });
        stdout.printf ("Initial stock is %d\n", store.getStock());
        stdout.printf ("Initial price is $ %f\n", store.getPrice());
        store.removeStock(4);

        store.setPrice(5.0);
        stdout.printf ("Stock is %d\n", store.getStock());
        stdout.printf ("and price is now $ %f\n", store.getPrice());

        store.removeStock(4);

        var status = "still available";
        if (store.isAvailable() == false) {
            status = "not available";
        }
        stdout.printf ("And the book is %s\n", status);
        store.setPrice(0.2);
```

4. 위를 실행하라.
5. 출력된 메시지를 확인하라.

```
Hello, world
1234
Author name: Joe Random
Author name: Joe Random Jr.
Initial stock is 10
Initial price is $ 4.200000
Stock is 6
and price is now $ 5.000000
Uh oh, we are going to run out stock soon!
And the book is still available
Uh oh, price is too low
```

무슨 일이 일어났는가**?**

스톡이 실행되고 있으며 가격이 너무 낮다고 출력된 경고 메시지는 BookStore 클래스가 출력하는 것이 아니라 Main 클래스에 의해 출력된다. 이는 Main 클래스가 시그널을 구독하는 시나리오를 가정하며, Main 클래스가 시그널로부터 정보를 수신하면 무언가 조치를 취할 것이다.

```
public signal void stockAlert();
public signal void priceAlert();
```

먼저 시그널을 발행하려는 클래스에서 시그널을 정의해야 한다. BookStore에서 우리는 두 개의 시그널을 선언한다. signal 키워드로 된 메서드 시그니처만 선언함을 주목한다. 따라서 함수의 본체는 선언하지 않는다. 이러한 시그널을 구독하는 객체는 발생된 시그널을 처리하도록 함수를 제공한다는 사실은 매우 중요하다.

```
    if (stock < 5) {
        stockAlert();
    }
...
    if (price < 1) {
        priceAlert();
    }
```

두 개의 코드 조각은 시그널을 어떻게 발생시키는지 보여준다. stock이 5보다 적으면 stockAlert 시그널을, price가 1보다 적으면 priceAlert 시그널을 발생시킨다. BookStore 클래스는 다음에 발생하는 일은 신경 쓰지 않고 시그널에 대해 알릴(announce) 뿐이다.

```
store.stockAlert.connect(() => {
    stdout.printf ("Uh oh, we are going to run out stock soon!\n");
});
store.priceAlert.connect(() => {
    stdout.printf ("Uh oh, price is too low\n");
});
```

여기서는 Main 클래스 구조체가 스스로를 두 개의 시그널로 연결한다. => 연산자를 이용하면 함수 본체를 제공하기 위한 구조체를 확인할 수 있다. 이러한 구조체를 클로저(closure) 또는 익명 함수(anonymous function)라고 부른다. 해당 함수의 매개변수는 => 앞에 정의되므로, 현재 맥락이라면 어떤 매개변수도 제공되지 않았음을 가리킬 것이다. 이러한 사실은 빈 괄호 내용으로 알 수 있다.

함수 본체 내에서 우리는 store 객체가 시그널을 발생시키면 무슨 일이 일어나야 하는지를 선언한다. 본문에서는 일부 경고 텍스트만 출력한다. 실제로는 네트워크의 연결을 해제하고 이미지를 표시하는 행위를 비롯해 원하는 액션을 모두 실행할 수 있다.

```
store.removeStock(4);
...
store.setPrice(0.2);
```

이제 실제 시그널이 발생하고 텍스트가 출력된다.

시도해보기 - 매개변수를 시그널에 넣기

매개변수를 시그널에 넣을 수도 있다. 그저 원하는 매개변수를 시그널 선언에 넣으면 된다. 그리고 시그널로 연결할 때 => 연산자 앞에 매개변수를 넣는다. 그렇다면 priceAlert 시그널이 하나의 매개변수, 즉 책의 가격을 매개변수로 갖도록 수정하는 건 어떨까?

요약

애플리케이션을 생성하고 곧바로 Seed와 Vala를 실행하는 작업은 쉬우면서도 빠르다. 그렇다면 이 책에서 둘 다 학습하고 사용하는 이유는 무얼까?

JavaScript는 해석형(interpreted) 언어로, 프로그램의 내용물을 확인하고 재컴파일이 필요 없이 직접 수정이 가능하다. 반면 Vala는 컴파일러형 언어에 해당한다. 소스 코드를 수정하기 위해서는 그곳에 접근할 필요가 있다. GNOME 플랫폼에 상업용 소프트웨어를 만들고자 한다면 Vala가 더 나은 선택이다.

Seed에서 JavaScript로 프로그램을 만들기란 꽤 쉬우며 Anjuta에서 프로젝트 관리를 필요로 하지 않는 반면 Vala에서는 수동으로 종속성을 처리해야 한다. 향후 Anjuta 버전에서는 이 문제가 해결되길 바란다.

이제 JavaScript와 Vala 코드의 기본적인 구조체, 즉 기본 데이터 유형을 조작하는 것부터 객체 지향 프로그래밍 개념을 사용하는 것까지 이해하게 되었다.

JavaScript 프로그래밍은 꽤 느슨한 반면 Vala는 엄격하다. 모듈화를 이용한 더 나은 코드 구조체를 활용하면 개발을 단순화하고 디버깅을 수월하게 만드는 데 도움이 될 것이다.

이 모든 내용을 이해했으니 다음 장에서 학습할 GNOME 애플리케이션 생성의 기반이 되는 GNOME 플랫폼 라이브러리를 사용할 준비가 된 셈이다.

Notes

GNOME3ApplicationDevelopmentBeginnersGuide:Chap 04

제 4 장 GNOME 코어 라이브러리 사용하기

GNOME 코어 라이브러리 사용하기

GNOME 코어 라이브러리는 기본 유틸리티 클래스와 함수의 집합체다. 이는 간단한 일정 변환 함수부터 가상 파일시스템 접근 관리까지 많은 일을 다룬다. 코어 라이브러리가 없었다면 *GNOME*은 지금처럼 강력하지 않았을 것이다. 세상에는 이러한 강력한 기능이 없어 성공하지 못한 *UI* 라이브러리들이 많다. 그러한 기능들을 지원하기 위해 *GNOME* 코어 라이브러리를 사용하는 라이브러리가 *GNOME* 외에도 많다는 사실은 놀라운 일이 아니다.

GNOME 코어 라이브러리는 UI 애플리케이션을 지원하기 위한 non-UI 라이브러리인 GLib과 GIO로부터 구성된다. 해당 라이브러리들은 파일, 네트워크, 타이머, 운영체제 내 다른 중요한 측면들과 프로그램을 연결한다. 이러한 지식이 없다면 아름다운 프로그램은 만들 수 있을지언정 나머지 시스템과 상호작용을 할 수 없을지도 모른다.

이번 장에서 학습할 내용은 다음과 같다.

- GLib 메인 루프와 기본 함수
- GObject 시그널링 시스템과 프로퍼티
- GIO 파일, 스트림, 네트워킹
- GSettings 설정 시스템

그럼 시작해보자.

시작하기 전에

이번 장에는 Internet이나 로컬 네트워크로 접근을 요하는 연습문제가 몇 개 실려 있다. 프로그램을 실행하기 전에 먼저 Internet이나 로컬 네트워크로 양호하게 연결되어 있는지 확인하라. 제거가능(removable) 하드웨어와 탑재 가능(mountable) 파일시스템을 필요로 하는 연습문제도 실려있다.

이번 장에서는 Vala 연습문제에 대해 좀 더 다른 방식으로 시도해볼 것이다. 논의의 특성은 서로 무관하기 때문에 하나의 프로젝트에서 계속해서 파일을 수정하는 대신 Vala의 각 연습문제를 고유의 프로젝트에서 진행한다. 따라서 Vala 연습문제마다 새로운 프로젝트를 생성하고 그 프로젝트 내에서 작업할 것이다. 본 서적에 동봉된 소스 코드와 자신의 프로젝트를 쉽게 비교할 수 있도록 각 프로젝트의 이름을 명시하겠다. 앞 장과 마찬가지로 본문에 생성된 프로젝트는 Vala GTK+(간단한) 프로젝트다. 프로젝트 프로퍼티에서 **GtkBuilder support for user interface** 옵션을 체크해제하고 **License** 옵션에서 **No licence** 항목을 선택한다.

각 연습문제에서 JavaScript 코드는 Vala 코드를 따르고, 각 연습문제는 하나의 파일로 저장된다. JavaScript 코드의 기능은 정확히 같을 것이다. 따라서 Vala와 JavaScript 코드 중 하나를 이용하거나 둘 다 이용해도 좋다.

GLib 메인 루프

GLib는 메인 이벤트 루프로서 다양한 소스로부터 들어오는 이벤트를 처리한다. 이러한 이벤트 루프를 이용하면 이벤트를 포착하여 필요한 처리를 실행할 수 있다.

실행하기 - **GLib** 메인 루프 갖고 놀기

GLib 메인 루프를 소개하겠다.

1. core-mainloop라는 새로운 Vala 프로젝트를 생성하고 Main 클래스에 아래 코드를 이용하라.

```
using GLib;

public class Main : Object
{
    int counter = 0;

    bool printCounter() {
        stdout.printf("%d\n", counter++);
        return true;
    }

    public Main ()
    {
        Timeout.add(1000, printCounter);
    }

    static int main (string[] args)
    {
        Main main = new Main();
        var loop = new MainLoop();
        loop.run ();
        return 0;
    }
}
```

2. 그리고 아래는 JavaScript 코드에 해당하는 내용으로, 스크립트를 core-mainloop.js로 명명할 수 있다.

```
#!/usr/bin/env seed

GLib = imports.gi.GLib;
GObject = imports.gi.GObject;

Main = new GType({
    parent: GObject.Object.type,
    name: "Main",
    init: function() {
        var counter = 0;
        this.printCounter = function() {
            Seed.printf("%d", counter++);
            return true;
        };
        GLib.timeout_add(0, 1000, this.printCounter);
    }
});
var main = new Main();
var context = GLib.main_context_default();
var loop = new GLib.MainLoop.c_new(context);
loop.run();
```

3. 프로그램을 실행하라. 프로그램이 카운터를 출력하고 계속 실행 중임을 확인할 수 있는가? 실행을 중지하는 유일한 방법은 Ctrl+C 키를 누르는 것이다.

무슨 일이 일어났는가?

이벤트의 유일한 소스인 timeout으로 GLib 메인 루프를 설정하였다.

먼저 counter 변수를 0으로 설정한다.

```
int counter = 0;
```

counter 변수의 값을 출력하기 위해 printCounter 라는 함수를 준비하고, 값을 출력한 직후 그 값을 증가시킨다. 이후 true를 리턴하여 카운터가 계속 계수하도록 한다.

```
bool printCounter() {
    stdout.printf("%d\n", counter++);
    return true;
}
```

생성자 내에서 1000 ms 간격으로 printCounter 함수를 가리키는 Timeout 객체를 생성한다. 즉, 1 초 간격으로 printCounter가 호출되고, printCounter가 true를 리턴하는 한 계속 반복하여 호출될 것이라는 의미다.

```
public Main ()
{
    Timeout.add(1000, printCounter);
}
```

main 함수에서 우리는 Main 클래스를 인스턴스화하고, MainLoop 객체를 생성하며, run을 호출한다. 이에 따라 프로그램을 수동으로 종료할 때까지 계속 실행될 것이다. 루프가 실행되면 루프로 전송된 이벤트를 수락한다. 앞에서 생성하였던 Timeout 객체가 그러한 이벤트를 생성한다. 타이머 간격이 만료될 때마다 메인 루프

에게 그 사실을 알리면 메인 루프는 printCounter 함수를 호출한다.

```
static int main (string[] args)
{
    Main main = new Main();
    var loop = new MainLoop();
    loop.run ();
    return 0;
}
```

이제 JavaScript 코드를 살펴보자. 눈치 챈 사람이 있을지 모르지만 클래스 구조체가 앞 장에서 학습한 것과 약간 다르다. 여기서는 Seed Runtime의 클래스 구조를 사용했다.

```
GLib = imports.gi.GLib;
GObject = imports.gi.GObject;
```

여기서 GLib와 GObject를 가져온다. 다음으로 GObject를 기반으로 하는 Main이라는 클래스를 생성한다.

우리가 할 일은 이렇다. 아래 코드는 GType 을 Main이라 불리는 클래스로 상속(subclass)하고 객체 구조체를 인자로 전달함을 나타낸다.

```
Main = new GType({
    parent: GObject.Object.type,
    name: "Main",
```

객체의 첫 번째 member는 parent로, 우리 클래스의 부모 클래스다. 해당 클래스에는 GObject.Object.type 을 할당하여 클래스가 앞서 가져온 GObject 모듈의 Object로부터 파생됨을 표시한다. 그리고 클래스를 Main이라고 명명한다. 이후 함수를 클래스 생성자이기도 한 init 함수에 넣는다. 클래스 member의 내용은 Vala 코드에서 살펴본 것과 비슷하며 꽤 간단하다.

```
    init: function() {
        var counter = 0;
        this.printCounter = function() {
            Seed.printf("%d", counter++);
            return true;
        };
        GLib.timeout_add(0, 1000, this.printCounter);
    }
});
```

그러면 Vala의 정적인 main 함수에 포함된 것과 동일한 코드가 된다. 여기서 Main 객체를 생성하고 GLib의 메인 루프를 생성한다.

```
var main = new Main();
var context = GLib.main_context_default();
var loop = new GLib.MainLoop.c_new(context);
loop.run();
```

시도해보기 - Timeout 중지하기

우리 프로그램은 끝이 없이 계수를 지속한다. 카운터가 10까지 도달하면 계수를 중지할 수 있는가?

[	printCounter 리턴값을 이용하면 된다.	]

아니면 카운터가 10에 도달하면 프로그램이 종료되도록 계수를 완전히 중지시킬 수 있는가?

[	리턴값을 무시하고 코드를 재배열하며 어떻게서든 loop 객체를 Main 클래스로 전달하는 것도 가능하다. printCounter 함수에서 loop.quit()을 호출하면 10에 도달할 때마다 프로그램이 프로그램적으로 메인 루프를 깨도록 만들 수 있다.	]

GObject 시그널

GObject는 우리가 소개할 수 있는 시그널링 메커니즘을 제공한다. 앞 장에서는 Vala 시그널링 시스템에 대해 논했다. 내부적으로는 사실상 GObject 시그널링 시스템을 사용하지만 언어 자체로 균일하게 통합되었음은 분명하다.

실행하기 - GObject 시그널 처리하기

JavaScript에서 이를 어떻게 실행하는지 살펴보자.

1. core-signals.js라고 불리는 새로운 스크립트를 생성하고 아래 코드로 채워라.

```
#!/usr/bin/env seed

GLib = imports.gi.GLib;
GObject = imports.gi.GObject;

Main = new GType({
    parent: GObject.Object.type,
    name: "Main",
    signals: [
            {
                name: "alert",
                parameters: [GObject.TYPE_INT]
            }
        ],
    init: function(self) {
        var counter = 0;

        this.printCounter = function() {
            Seed.printf("%d", counter++);
            if (counter > 9) {
                self.signal.alert.emit(counter);
            }
            return true;
        };

        GLib.timeout_add(0, 1000, this.printCounter);
```

```
    }
});

var main = new Main();

var context = GLib.main_context_default();
var loop = new GLib.MainLoop.c_new(context);
main.signal.connect('alert', function(object, counter) {
    Seed.printf("Counter is %d, let's stop here", counter);
    loop.quit();
});
loop.run();
```

2. 스크립트를 실행하고 출력된 메시지를 확인하라.

```
0
1
2
3
4
5
6
7
8
9
Counter is 10, let's stop here
```

무슨 일이 일어났는가?

GObject 시그널링 시스템을 이용하면 객체가 발생시키는 알림을 구독할 수 있다. 단지 시그널을 수신하면 특정 액션을 실행하는 핸들러를 제공하기만 하면 된다. 여기서 우리는 이름과 매개변수가 있는 객체를 객체의 내용으로 넣음으로써 배열에서 시그널을 선언한다. 매개변수 타입은 GLib 시스템이 알고 있는 타입이다. 시그널에 어떤 매개변수도 포함하지 않은 경우 생략해도 좋다.

```
signals: [
        {
            name: "alert",
            parameters: [GObject.TYPE_INT]
        }
    ],
main.signal.connect('alert', function(object, counter) {
    Seed.printf("Counter is %d, let's stop here", counter);
    loop.quit();
});
```

이후 시그널을 구독하고, counter 값을 출력하고 메인 루프를 깨는 클로저를 제공한다. 매개변수는 클로저의 두 번째 매개변수에서 정의됨을 주목하라. 첫 번째 매개변수는 객체 자체를 위해 예약(reserve)된다.

마지막으로 시그널의 이름을 호출함으로써 시그널을 발생시킨다. self는 init 함수에서 전달하는 Main 클래스다.

```
if (counter > 9) {
    self.signal.alert.emit(counter);
}
```

위를 호출하자마자 시그널은 메인 루프에서 처리되고 그것을 구독하는 객체로 전달될 것이다.

시도해보기 - Vala에서 작성하기

마지막에서 살펴보았듯이 앞의 코드와 비교할 때 시그널 선언, 시그널 발생, 구독은 Vala에서 더 수월하다. 앞의 코드를 Vala에서 쓴다면 어떤가?

GLib 프로퍼티

프로퍼티는 저장공간 시스템에서 키-값 쌍으로, GNOME 시스템의 모든 객체에 기반 클래스인 GObject의 모든 인스턴스에서 이용할 수 있다. 프로퍼티의 유용한 기능으로, 값이 변경되면 변경 내용을 구독할 수 있다는 점을 들 수 있겠다.

실행하기 - 프로퍼티 접근하기

프로퍼티 값이 변경되는지 감시할 뿐만 아니라 프로퍼티로/로부터 값을 설정하고 얻는 방법을 학습할 것이다.

1. core-properties.js라는 새로운 스크립트를 생성하고 아래 코드 내용으로 채워라.

```
#!/usr/bin/env seed

GLib = imports.gi.GLib;
GObject = imports.gi.GObject;

Main = new GType({
    parent: GObject.Object.type,
    name: "Main",
    properties: [
        {
            name: 'counter',
            type: GObject.TYPE_INT,
            default_value: 0,
            minimum_value: 0,
            maximum_value: 1024,
            flags: (GObject.ParamFlags.CONSTRUCT
                | GObject.ParamFlags.READABLE
                | GObject.ParamFlags.WRITABLE),
        }
    ],
    init: function(self) {
        this.print_counter = function() {
            Seed.printf("%d", self.counter++);
            return true;
        }

        this.monitor_counter = function(obj, gobject, data) {
            Seed.print("Counter value has changed to " + obj.counter);
        }
```

```
        GLib.timeout_add(0, 1000, this.print_counter);
        self.signal.connect("notify::counter", this.monitor_counter);
    }
});

var main = new Main();
var context = GLib.main_context_default();
var loop = new GLib.MainLoop.c_new(context);
loop.run();
```

2. Vala에 해당하는 내용은 다음과 같다 (core-properties라는 새로운 프로젝트를 생성하고 아래 코드로 core_properties.vala를 채워라).

```
using GLib;

public class Main : Object
{
    public int counter {
        set construct;
        get;
        default = 0;
    }

    public bool print_counter() {
        stdout.printf("%d\n", counter ++);
        return true;
    }

    public void monitor_counter() {
        stdout.printf ("Counter value has changed to %d\n", counter);
    }

    public Main ()
    {
    }

    construct {
        Timeout.add(1000, print_counter);
        notify["counter"].connect ((obj)=> {
            monitor_counter ();
        });
    }

    static int main (string[] args)
    {
        Gtk.init (ref args);
        var app = new Main ();
```

```
        Gtk.main ();
        return 0;
    }
}
```

3. 위를 실행하고 출력되는 메시지를 확인하라. Ctrl+C 를 누르면 프로그램이 중지된다는 사실을 기억한다.

```
Counter value has changed to 0
Counter value has changed to 1
0
Counter value has changed to 2
1
Counter value has changed to 3
2
Counter value has changed to 4
3
Counter value has changed to 5
4
Counter value has changed to 6
5..
```

무슨 일이 일어났는가?

JavaScript 코드에서는 properties 배열 안에서 프로퍼티를 선언하고, 프로퍼티의 객체로 내용을 채워야 한다.

여기서 우리는 프로퍼티가 counter라는 이름을 갖고 있으며 정수 타입임을 설명한다. 기본값, 최소값, 최대값을 선언할 필요가 있다. 플래그 또한 필요로 한다. 플래그로부터 GObject.ParamFlags.CONSTRUCT를 확인할 수 있는데, 이는 프로퍼티가 생성 단계(construction phase)에서 초기화됨을 의미한다. 즉, 객체가 생성될 때 기본값이 설정된다는 뜻이다. 또 읽고 쓰기가 가능함을 확인할 수 있다.

```
properties: [
        {
            name: 'counter',
            type: GObject.TYPE_INT,
            default_value: 0,
            minimum_value: 0,
            maximum_value: 1024,
            flags: (GObject.ParamFlags.CONSTRUCT
                | GObject.ParamFlags.READABLE
                | GObject.ParamFlags.WRITABLE),
        }
    ]
```

아래 코드를 통해 우리는 변경되는 내용을 구독할 수 있다. 시그널링 시스템을 이용하며, 시그널의 이름은 notify:: 키워드 다음에 프로퍼티명을 이용해 생성된다. 그 다음으로 프로퍼티가 변경될 때마다 시그널 핸들러를 트리거할 것이다.

```
self.signal.connect("notify::counter", this.monitor_counter);
```

여기서 이제 프로퍼티의 값을 증가시킴으로써 값을 설정한다. 이 때 값을 수정하기 때문에 value monitor가 먼저 트리거된 후에 실제 값이 printf에 의해 출력됨을 주목한다.

```
this.print_counter = function() {
    Seed.printf("%d", self.counter++);
    return true;
}
```

아래 코드는 값을 어떻게 읽는지를 보여준다.

```
this.monitor_counter = function(obj, gobject, data) {
    Seed.print("Counter value has changed to " + obj.counter);
}
```

JavaScript 코드와 반대로 Vala에서 프로퍼티의 선언은 매우 간단하다. 선언은 몇 가지가 추가된 일반적인 변수 선언과 비슷하다.

```
public int counter {
    set construct;
    get;
    default = 0;
}
```

하지만 최소값과 최대값을 설정하기 위한 메커니즘이 없다.

그 다음, 일반 변수를 읽고 쓰는 것과 마찬가지로 프로퍼티의 읽고 쓰기가 이루어짐을 확인할 수 있다. 클래스 외부에서는 member 변수를 참조하기 위해 일반적인 방법, 즉 객체명 다음에 마침표와 프로퍼티명을 붙여 이용할 수 있다.

```
public bool print_counter() {
    stdout.printf("%d\n", counter ++);
    return true;
}

public void monitor_counter() {
    stdout.printf ("Counter value has changed to %d\n", counter);
}
```

변경내용의 구독 또한 일반적인 시그널링 메커니즘을 사용하는데, 단, 시그널명 notify 다음에 사각 괄호에 프로퍼티명을 삽입하는 경우는 제외다.

```
notify["counter"].connect ((obj)=> {
    monitor_counter ();
}
```

코드에 이전에 보지 못한 새로운 내용이 눈에 띈다. 바로 construct 키워드다. 이것은 일반적인 생성자와 비슷한 객체를 생성하는 또 다른 방법이다. 이러한 생성 스타일은 실제 생성된 C 코드에서 GObject 생성이 실행되는 방법에 가깝다.

JavaScript와 Vala 코드의 이러한 차이에도 불구하고 둘 다 여느 클래스의 member와 똑같이 프로퍼티의 사용을 허용한다. 따라서 두 언어에서 main.counter로서 counter 프로퍼티로 접근이 가능하다 (객체명이 main이라고 가정하고).

팝 퀴즈 - 왜 0 값이 출력되는가

출력된 내용을 확인하면 다음과 같다.

```
Counter value has changed to 0
```

Q1. 카운터를 **0**으로 명시적으로 설정하지 않았다**.** 왜 이런 일이 발생했는가**?**

1. 프로퍼티에 set construct 키워드가 정의되어 있기 때문이다.
2. 0이 기본값이기 때문이다.

시도해보기 - 프로퍼티를 읽기 전용으로 만들기

프로퍼티가 읽기 전용이라면 더 이상 그 값을 설정할 수가 없다. 이제 counter 프로퍼티를 읽기 전용으로 만들어보자. 힌트: 프로퍼티 플래그를 활용하라.

설정 파일

많은 상황에서 우리는 프로그램이 행동하는 방식을 맞춤설정하기 위해 어떻게든 설정 파일에서 읽어와야 한다. 지금부터는 설정 파일을 이용해 GLib에서 가장 단순한 설정 메커니즘을 사용하는 방법을 학습할 것이다. 먼저 설정 파일이 있는데, 그 파일에는 애플리케이션의 이름과 버전이 포함되어 있어 프로그램 내에 어딘가에 출력할 수 있다고 가정해보자.

실행하기 - 설정 파일 읽기

우리가 할 일은 다음과 같다.

1. 설정 파일을 생성하고 core-keyfile.init라고 부르자. 그 내용은 다음과 같다.

```
[General]
name = "This is name"
version = 1
```

2. 새로운 Vala 프로젝트를 생성하고 core-keyfile로 명명하라. (src가 아니라) 프로젝트 디렉터리 안에 core-keyfile.ini 파일을 넣어라.

3. core_keyfile.vala가 아래와 같은 모습이 되도록 편집하라.

```
using GLib;

public class Main : Object
{
    KeyFile keyFile = null;
    public Main ()
    {
        keyFile = new KeyFile();
        keyFile.load_from_file("core-keyfile.ini", 0);
    }

    public int get_version()
    {
        return keyFile.get_integer("General", "version");
    }

    public string get_name()
    {
        return keyFile.get_string("General", "name");
```

```
    }

    static int main (string[] args)
    {
        var app = new Main ();
        stdout.printf("%s %d\n", app.get_name(), app.get_version());

        return 0;
    }
}
```

4. JavaScript 코드(core-keyfile.js라고 부르자)는 다음과 같은 모습이다 (.ini 파일은 스크립트와 동일한 디렉터리에 넣을 것을 기억하라).

```
#!/usr/bin/env seed

GLib = imports.gi.GLib;
GObject = imports.gi.GObject;

Main = new GType({
    parent: GObject.Object.type,
    name: "Main",
    init: function(self) {

        this.get_name = function() {
            return this.keyFile.get_string("General", "name");
        }

        this.get_version = function() {
            return this.keyFile.get_integer("General", "version");
        }
        this.keyFile = new GLib.KeyFile.c_new();
        this.keyFile.load_from_file("core-keyfile.ini");
    }
});
var main = new Main();
Seed.printf("%s %d", main.get_name(), main.get_version());
```

5. 프로그램을 실행하고 출력된 내용을 확인하라.

```
"This is name" 1
```

무슨 일이 일어났는가?

우리가 사용 중인 설정 파일은 freedesktop.org의 Desktop Entry Specification 문서를 준수하는 키-값 쌍 구조체를 갖고 있다. GNOME 플랫폼에서는 이 구조체가 흔히 사용되는데, 런처에 의해 사용되는 .desktop 파일에서 활용되는 것이 보통이다. Windows를 사용하는 사람이라면 .ini 포맷과 비슷함을 발견할 것인데 이 포맷도 마찬가지로 설정에 사용된다.

GLib는 이러한 타입의 설정 파일에 접근하기 위해 KeyFile 클래스를 제공한다. 생성자에 다음과 같은 코드 조각이 발견될 것이다.

```
keyFile = new KeyFile();
keyFile.load_from_file("core-keyfile.ini", 0);
```

이는 KeyFile의 객체를 초기화하고 core-keyfile.ini 파일을 객체로 로딩한다.

core-keyfile.ini 파일을 잠시 살펴보면, 한 쌍의 사각 괄호 안에 섹션이 쓰여 있다.

```
[General]
```

그 다음에 따라오는 모든 엔트리는 섹션명을 명시함으로써 접근 가능하다. 여기서는 두 가지 메서드, get_version()과 get_name()을 제공하는데, 이는 설정 파일에서 name과 version 엔트리의 값을 얻는 데에 사용되는 단축키에 해당한다.

```
public int get_version()
{
    return keyFile.get_integer("General", "version");
}

public string get_name()
{
    return keyFile.get_string("General", "name");
}
```

메서드 내부를 보면 version 엔트리에서 정수값을 얻고 name 엔트리에서 문자열 값을 얻을 뿐이다. General 섹션으로부터 엔트리를 얻음을 확인할 수도 있다. 이러한 메서드 내에서는 값을 즉시 리턴한다.

아래 코드에서 보여주듯이 우리는 메서드로부터 값을 소모하고 출력한다.

```
stdout.printf("%s %d\n", app.get_name(), app.get_version());
```

꽤 쉽지 않은가? JavaScript 코드는 쉽고 간단하기 때문에 더 이상 설명할 필요가 없다.

시도해보기 - 다중 섹션 설정

설정 파일 내부에 더 많은 섹션을 추가하고 값으로 접근해보자. 가령 license_file과 customer_id를 엔트리로 갖고 있는 License라는 특정 섹션이 있다고 치자. 후에 이 정보를 이용해 고객이 소프트웨어를 이용할 권리를 갖고 있는지 확인한다고 가정하자.

GIO, 입/출력 라이브러리

실제 세계에서 우리 프로그램은 로컬로 저장되든 원격으로 저장되든 파일로 접근할 수 있어야 한다. 읽어야 할 파일의 집합이 있다고 가정해보자. 파일은 로컬로, 그리고 원격으로 퍼져있다. GIO는 추상적인 방식으로 파일과 상호작용하도록 API를 제공하기 때문에 이러한 파일을 쉽게 조작하도록 해준다.

실행하기 - 파일 접근하기

어떻게 작용하는지 살펴보자.

1. core-files.js라는 새로운 스크립트를 생성하고 아래의 행으로 채워라.

```
#!/usr/bin/env seed

GLib = imports.gi.GLib;
Gio = imports.gi.Gio;
GObject = imports.gi.GObject;

Main = new GType({
    parent: GObject.Object.type,
    name: "Main",
    init: function(self) {
        this.start = function() {
            var file = null;
            var files = ["http://en.wikipedia.org/wiki/Text_file",
"core-files.js"];
            for (var i = 0; i < files.length; i++) {
                if (files[i].match(/^http:/)) {
                    file = Gio.file_new_for_uri(files[i]);
                } else {
                    file = Gio.file_new_for_path(files[i]);
                }

                var stream = file.read();
                var data_stream = new
Gio.DataInputStream.c_new(stream);
                    var data = data_stream.read_until("", 0);

                    Seed.print(data)
            }
        }
    }
});

var main = new Main();
main.start();
```

2. 아니면 core-files라는 Vala 프로젝트를 생성할 수 있다. src/core_files.vala를 아래의 코드로 채워라.

```
using GLib;

public class Main : Object
```

```
{
    public Main ()
    {
    }

    public void start ()
    {
        File file = null;
        string[] files = {"http://en.wikipedia.org/wiki/Text_file",
"src/core_files.vala"};

        for (var i = 0; i < files.length; i++) {
            if (files[i].has_prefix("http:")) {
                file = File.new_for_uri(files[i]);
            } else {
                file = File.new_for_path(files[i]);
            }

            var stream = file.read();
            var data_stream = new DataInputStream(stream);
            size_t data_read;
                var data = data_stream.read_until("", out data_read);
                stdout.printf(data);
        }
    }

    static int main (string[] args)
    {
        var app = new Main ();
        app.start();
        return 0;
    }
}
```

3. 프로그램을 실행하고, Internet에서 Wikipedia 페이지와 함께 로컬 디렉터리에서 프로그램의 소스 코드를 가져옴을 주목하라.

무슨 일이 일어났는가**?**

GIO는 강력한 가상 파일시스템 API 세트를 제공한다. 인터페이스 세트를 제공하여 특정 구현에 의해 확장되는 기반의 역할을 한다. 예를 들어, 여기서는 파일에 대한 함수를 정의하는 GFile 인터페이스를 사용한다. GFile API는 파일이 어디에 위치하는지, 파일이 어떻게 읽히는지 또는 다른 세부 사항은 알려주지 않는다. 그저 함수만 제공하며 그 뿐이다. 애플리케이션 개발자에게 주어지는 구체적인 구현이 사실상 모든 일을 수행한다. 이것이 무슨 뜻인지 살펴보자.

아래 코드에서는 files 배열로부터 파일 위치를 얻는다. 그리고 위치에 HTTP 프로토콜 식별자가 있는지를 확인하며, 식별자가 발견되면 file_new_for_uri를 이용해 GFile 객체를 생성하고, 식별자가 없으면 file_new_for_path를 이용한다. 로컬 파일에도 물론 file_new_for_uri를 사용할 수 있지만 파일명 뒤에 file:// 프로토콜 식별자를 붙여야 한다.

```
if (files[i].match(/^http:/)) {
    file = Gio.file_new_for_uri(files[i]);
} else {
    file = Gio.file_new_for_path(files[i]);
}
```

이는 원격 파일의 처리와 로컬 파일의 처리의 유일한 차이점이다. 이후부터는 GIO와 동일한 함수를 이용해 로컬 드라이브 또는 웹 서버로부터 파일로 접근할 수 있다.

```
var stream = file.read();
var data_stream = new Gio.DataInputStream.c_new(stream);
var data = data_stream.read_until("", 0);
```

여기서 GFileInputStream 객체를 얻기 위해 read 함수를 이용한다. API는 파일이 어디 있든 동일한 함수를 제공함을 명심한다.

결과가 되는 객체는 스트림이다. 스트림은 한 곳에서 다른 곳으로 흐르는 데이터의 시퀀스다. 스트림은 객체로 전달되어 다른 스트림이 되도록 변형하거나 소모할 수 있다.

본문에 소개된 예제의 경우 처음엔 file.read 함수로부터 스트림을 얻는다. 데이터를 쉽게 읽도록 이 스트림을 GDataInputStream으로 전송한다. 새로운 스트림을 이용해 어떤 내용도 찾지 않을 때까지, 즉 파일 끝에 도달할 때까지 데이터를 읽을 것을 GIO에게 요청한다. 이후 데이터를 화면으로 출력한다.

GIO를 이용한 네트워크 접근

GIO는 네트워크로 접근하는 데에 적절한 함수를 제공한다. 이번 절에서는 소켓 클라이언트와 서버 프로그램을 생성하는 방법을 학습할 것이다. 한 곳에서 다른 곳으로 데이터를 전송할 수 있는 간단한 채팅(chat) 프로그램을 만든다고 가정해보자.

실행하기 - 네트워크 접근하기

간략하게 JavaScript로만 실행하지만, 본 서적에 첨부된 코어 서버와 코어-클라이언트 프로젝트에서 Vala 프로그램을 살펴볼 수 있다. 그렇다면 네트워크로 접근하기 위해 어떤 단계들이 필요한지 살펴보도록 하자.

1. core-server.js라는 새로운 스크립트를 생성하여 아래의 내용으로 채워라.

```
#!/usr/bin/env seed

GLib = imports.gi.GLib;
Gio = imports.gi.Gio;
GObject = imports.gi.GObject;

Main = new GType({
    parent: GObject.Object.type,
    name: "Main",
    init: function(self) {
        this.process = function(connection) {
            var input = new Gio.DataInputStream.c_new
(connection.get_input_stream());
            var data = input.read_upto("\n", 1);
            Seed.print("data from client: " + data);
            var output = new Gio.DataOutputStream.c_new
(connection.get_output_stream());
            output.put_string(data.toUpperCase());
            output.put_string("\n");
            connection.get_output_stream().flush();
        }

        this.start = function() {
            var service = new Gio.SocketService();
            service.add_inet_port(9000, null);
            service.start();
            while (1) {
                var connection = service.accept(null);
                this.process(connection);
```

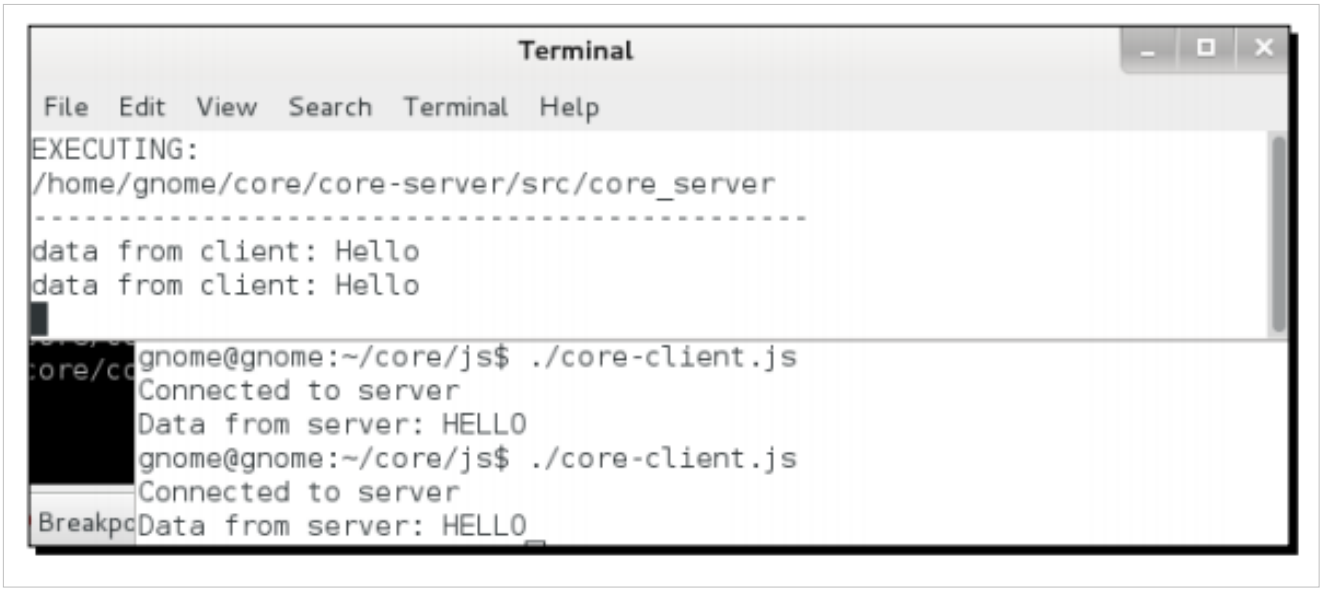
```
            }
        }
    }
});

var main = new Main();
main.start();
```

2. 스크립트를 실행하라. Ctrl+C를 직접 누를 때까지 프로그램은 실행될 것이다.

3. 다음으로 core-client.js라는 또 다른 스크립트를 생성하되 코드는 아래와 같다.

```
#!/usr/bin/env seed

GLib = imports.gi.GLib;
Gio = imports.gi.Gio;
GObject = imports.gi.GObject;

Main = new GType({
    parent: GObject.Object.type,
    name: "Main",
    init: function(self) {

        this.start = function() {
            var address = new Gio.InetAddress.from_string("127.0.0.1");
            var socket = new Gio.InetSocketAddress({address: address,
port: 9000});
            var client = new Gio.SocketClient ();
                var conn = client.connect (socket);

                Seed.printf("Connected to server");

            var output = conn.get_output_stream();
            var output_stream = new Gio.DataOutputStream.c_new(output);

                var message = "Hello\n";
            output_stream.put_string(message);
            output.flush();

            var input = conn.get_input_stream();
            var input_stream = new Gio.DataInputStream.c_new(input);
            var data = input_stream.read_upto("\n", 1);
            Seed.printf("Data from server: " + data);
        }
    }
});

var main = new Main();
main.start();
```

4. 프로그램을 실행하고, 서버와 클라이언트 프로그램의 출력 결과를 확인하라. 서로 대화를 할 수 있게 되었다!



무슨 일이 일어났는가?

GIO는 사용법이 매우 쉬운 저수준을 비롯해 고수준 네트워킹 API까지 제공한다. 서버를 먼저 살펴보자. 포트 번호 9000에 서비스를 연다. 이는 임의의 숫자로 원하는 숫자를 사용할 수도 있으나 몇 가지 수는 제한된다.

```
var service = new Gio.SocketService();
service.add_inet_port(9000, null);
service.start();
```

개발자의 포트 번호와 동일한 번호로 서비스가 실행 중이라면 또 다른 서비스를 실행할 수 없다. 1024 보다 적은 포트 번호를 사용하길 원한다면 프로그램을 루트로 실행해야 한다. 그리고 서비스가 들어오는 연결을 수락하면 호출되는 무한 루프로 들어간다. 여기서는 연결을 처리하기 위해 우리 프로세스 함수만 호출하면 끝이다.

```
while (1) {
    var connection = service.accept(null);
    this.process(connection);
}
```

서버의 기본 활동은 이처럼 쉽게 정의된다. 하지만 처리의 세부적인 내용은 또 다르다.

이후 연결에서 들어오는 입력 스트림을 바탕으로 GDataInputStream 객체를 생성한다. 그리고 행 문자의 끝, 즉 \n 문자를 찾을 때까지 데이터를 읽는다. 이것은 하나의 문자이므로 1을 넣어야 한다. 이후 들어오는 데이터를 출력한다.

```
var input = new Gio.DataInputStream.c_new
(connection.get_input_stream());
var data = input.read_upto("\n", 1);
Seed.print("data from client: " + data);
```

재미를 위해 클라이언트로 무언가 리턴하고자 한다. 연결 객체로부터 들어오는 GDataOutputStream 클래스의 객체를 생성한다. 클라이언트로부터 들어오는 데이터를 모두 대문자로 변경하고, 다시 스트림을 통해 전송한다. 마침내 파이프를 제거(flushing down the pipe)함으로써 모든 것이 전송되도록 확보한다. 서버측은 이게 모두다.

```
var output = new Gio.DataOutputStream.c_new
(connection.get_output_stream());
output.put_string(data.toUpperCase());
output.put_string("\n");
connection.get_output_stream().flush();
```

클라이언트측에서는 먼저 GInetAddress의 객체를 만든다. 이후 객체는 GInetSocketAddress로 입력되어 본인이 연결하길 원하는 주소의 포트를 정의할 수 있다.

```
var address = new Gio.InetAddress.from_string("127.0.0.1");
var socket = new Gio.InetSocketAddress({address: address, port: 9000});
```

다음으로 GSocketClient에서 socket 객체를 SocketClient와 연결한다. 이후 모든 것이 괜찮아지면 서버로 연결이 구축된다.

```
var client = new Gio.SocketClient ();
var conn = client.connect (socket);
```

클라이언트측에서는 대체적으로 서버측과 반대로 프로세스가 발생한다. 여기서는 연결 객체로부터 들어오는 스크림을 바탕으로 GDataOutputStream을 먼저 생성한다. 그 다음 이곳으로 메시지를 전송하기만 하면 된다. 파이프라인에 남은 데이터가 모두 제거되도록 flush 기능도 원할 것이다.

```
var output = conn.get_output_stream();
var output_stream = new Gio.DataOutputStream.c_new(output);

var message = "Hello\n";
output_stream.put_string(message);
output.flush();
```

서버로부터 무언가 얻을 것으로 예상하므로 입력 스트림 객체를 생성한다. 새로운 행을 찾을 때까지 읽어온 다음에 데이터를 출력한다.

```
var input = conn.get_input_stream();
var input_stream = new Gio.DataInputStream.c_new(input);
var data = input_stream.read_upto("\n", 1);
Seed.printf("Data from server: " + data);
```

시도해보기 - 에코 서버 만들기

에코 서버**(echo server)**는 그곳으로 전송된 모든 것을 어떠한 수정도 거치지 않고 본래 상태대로 리턴하는 서비스다. 가령, "Hello"라고 전송하면 서버는 "Hello"라고 전송한다. 때로는 두 개의 호스트 사이에 연결이 제대로 되는지 확인 시 사용된다. 서버 프로그램을 에코 서버로 수정하는 건 어떨까? 무한 루프로 넣을 수도 있지만 "quit"을 입력하면 서버의 연결은 해제된다.

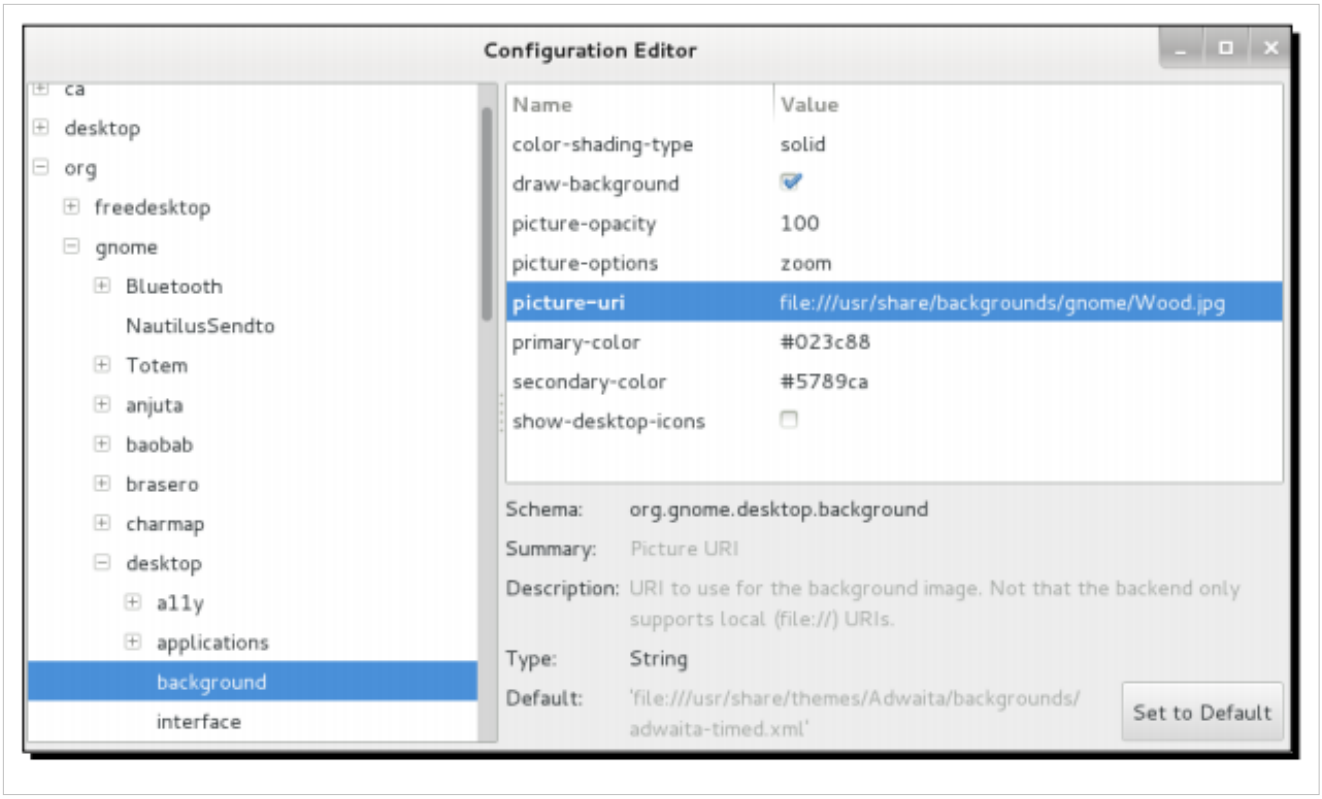
GSettings 이해하기

앞서 우리는 애플리케이션 설정을 읽기 위해 GLib 설정 파서(configuration parser)를 이용했다. 이제 GSettings를 이용해 좀 더 고급의 설정 시스템을 사용해보도록 하겠다. 이를 이용하면 시스템을 이용하는 모든 애플리케이션을 비롯해 GNOME 플랫폼에 걸쳐 설정(configurations)으로 접근할 수 있다.

실행하기 - GSettings 학습하기

dconf-editor 툴이 시각화한 GSettings 설정 시스템이 어떤 모습인지 살펴보자.

1. Terminal을 시작한다.
2. Terminal에서 dconf-editor를 실행한다.
3. 애플리케이션 왼쪽에 org 트리를 탐색하고, **gnome→desktop→background** 를 선택한다.



무슨 일이 일어났는가**?**

GSettings는 GNOME3에서 처음으로 도입되었다. 이전에는 GConf를 이용해 설정이 처리되었다. GNOME3에서는 포함된 모든 GNOME 애플리케이션이 GSettings를 사용하도록 이동되었다. 키-값 쌍을 이용해 설정을 GConf와 GSettings로 저장한다는 개념은 똑같이 남아 있다. 하지만 GSettings는 많은 측면에서 향상되었는데, 스키마를 메타데이터로 강요함으로써 사용이 좀 더 제한된다는 점도 포함된다. GConf를 이용하면 시스템에 자유롭게 값을 저장하고 그로부터 읽어올 수 있다.

GSettings는 사실상 유일한 최상위 수준 레이어다. 그 아래에는 dconf라고 불리는 저수준의 시스템이 있는데, 값을 실제로 저장하고 읽는 일을 처리한다. 여기서 논하는 툴은 키와 값을 상속구조로 표시하므로 값을 훑어보고 읽을 수 있으며 심지어 새 값을 쓸 수도 있다(물론 스키마가 새 값을 쓰는 것이 가능하다고 말할 경우).

스크린샷을 보면 org.gnome.desktop.background 에 다수의 엔트리가 있으며, 그 중 하나는 데스크톱 배경 이미지의 URI를 포함하는 picture-uri임을 확인할 수 있다.

GSettings API

이 서적에서는 API가 다른 관리 툴에 비해 흥미로운 툴에 해당한다. GSettings를 시각적으로 확인했다면 API를 통해 GSettings로 접근할 때가 되었다.

실행하기 - 프로그램적으로 GSettings에 접근하기

GNOME 데스크톱의 배경 이미지를 설정하기 위한 툴을 생성한다고 가정해보자. 다음과 같이 실행한다.

1. core-settings라는 새로운 Vala 프로젝트를 생성하고, core_settings.vala를 아래와 같이 수정하라.

```
using GLib;

public class Main : Object
{
    Settings settings = null;
    public Main ()
    {
        settings = new Settings("org.gnome.desktop.background");
    }

    public string get_bg()
    {
        if (settings == null) {
            return null;
        }

        return settings.get_string("picture-uri");
    }

    public void set_bg(string new_file)
    {
        if (settings == null) {
            return;
        }
        if (settings.set_string ("picture-uri", new_file)) {
            Settings.sync ();
        }
    }

    static int main (string[] args)
    {
        var app = new Main ();
        stdout.printf("%s\n", app.get_bg());
        app.set_bg ("file:///usr/share/backgrounds/gnome/Wood.jpg");
        return 0;
    }
}
```

2. JavaScript 코드는 꽤 간단하므로 아래에 일부만 소개하여 Vala 코드에서 어떤 적응(adaptation)을 필요로 하는지만 확인하도록 하겠다.

```
init: function(self) {
    this.settings = null;

    this.get_bg = function() {
        if (this.settings == null)
            return null;

        return this.settings.get_string("picture-uri");
    }

    this.set_bg = function(new_file) {
        if (this.settings == null)
            return;

        if (this.settings.set_string("picture-uri", new_file)) {
            Gio.Settings.sync();
        }
    }

    this.settings = new Gio.Settings({schema:
'org.gnome.desktop.background'});
}
```

3. 위를 실행하고 현재 데스크톱 배경 이미지에 변경된 내용을 확인하라. 현재 데스크톱 배경은 코드에 명시된 파일로 변경될 것이다.

무슨 일이 일어났는가**?**

이 연습문제에서는 데스크톱 소유로 이미 설치된 스키마, org.gnome.desktop.background를 사용하기 때문에 설정으로 접근하기 위해서는 API를 사용하면 된다. 자세한 내용을 살펴보자.

먼저 스키마 이름인 org.gnome.desktop.background를 명시함으로써 GSettings로 연결을 시작하면 GSettings 객체가 리턴된다.

```
settings = new Settings("org.gnome.desktop.background");
```

이후 초기화가 실패할 때를 대비해 간단한 안전망(safety net)을 넣는다. 실제 세계에서는 간단한 리턴 대신 재초기화를 실행할 수 있다.

```
if (settings == null) {
    return null;
}
```

이후 picture-uri 키 아래에서 string 타입의 값을 얻어 원하는 대로 소모할(consume) 수 있다.

```
return settings.get_string("picture-uri");
```

마지막으로 똑같은 키를 이용해 값을 설정한다. 설정이 성공하면 sync 함수를 호출함으로써 GSettings에게 그 값을 디스크로 저장하도록 요청한다. 쉽지 않은가?

```
if (settings.set_string ("picture-uri", new_file)) {
    Settings.sync ();
}
```

요약

이번 장에서는 GNOME 코어 라이브러리에 대해 많은 내용을 학습하였다. 라이브러리의 모든 내용을 다루진 않았지만 GNOME 애플리케이션을 빌드하는 데에 필요한 기본적이고 중요한 내용은 모두 살펴보았다.

이제 GLib는 다양한 소스로부터 이벤트를 처리하는 메인 루프를 제공한다는 사실을 알게 되었다. 그리고 GObject 프로퍼티와 시그널링 시스템에 관해 논했다. 또 timeout을 이용해 메인 루프가 처리하는 이벤트를 살펴보고, 프로퍼티의 값이 변경될 때 시그널도 살펴보았다. 프로그래밍 언어와 관련해, Vala는 GNOME과 좀 더 통합되고, JavaScript는 GObject 프로퍼티나 시그널을 사용하는 데에 더 많은 코드를 필요로 함을 알아냈다.

파일을 로컬과 원격으로 접근하는 실습도 해보고, GIO가 제공하는 API는 파일이 어디에 있든 접근하는 방식을 추상화하기 때문에 사용법이 매우 쉽다는 사실도 알아냈다.

GIO를 이용해 간단한 클라이언트 및 서버 채팅 프로그램을 만드는 연습도 실행했으며, 그러한 흥미로운 프로그램을 생성하기 위해서는 JavaScript와 Vala 모두 최소한의 코드량만 필요로 한다는 사실을 알아냈다.

마지막으로, GSettings에 관해 논하고 이를 이용해 GNOME 데스크톱 배경 이미지를 읽고 쓰는 연습을 했다.

GNOME 애플리케이션의 기본을 학습했으니 다음 장에서는 그래픽 프로그램의 기본 내용을 배워보겠다.

Notes

GNOME3ApplicationDevelopmentBeginnersGuide:Chapter 05

제 5 장 그래픽 사용자 인터페이스 애플리케이션 빌드하기

그래픽 사용자 인터페이스 애플리케이션 빌드하기

*GTK+*는 프로젝트가 시작될 때부터 사실상 *GNOME*을 위한 그래픽 사용자 인터페이스*(GUI)* 툴킷으로 사용되었다. 최근에는 *Clutter*가 대안책으로 통합되면서 *OpenGL*가 제공하는 유체 애니메이션 사용자 인터페이스를 제공하고 있다. 이 둘은 *GNOME*용 *GUI* 애플리케이션을 개발 시 선택할 수 있는 훌륭한 기본 툴킷 집합이다.

이번 장에서는 GTK+와 Clutter를 이용해 애플리케이션을 생성하는 방법을 학습할 것이다. 구체적으로 학습하게 될 내용은 다음과 같다.

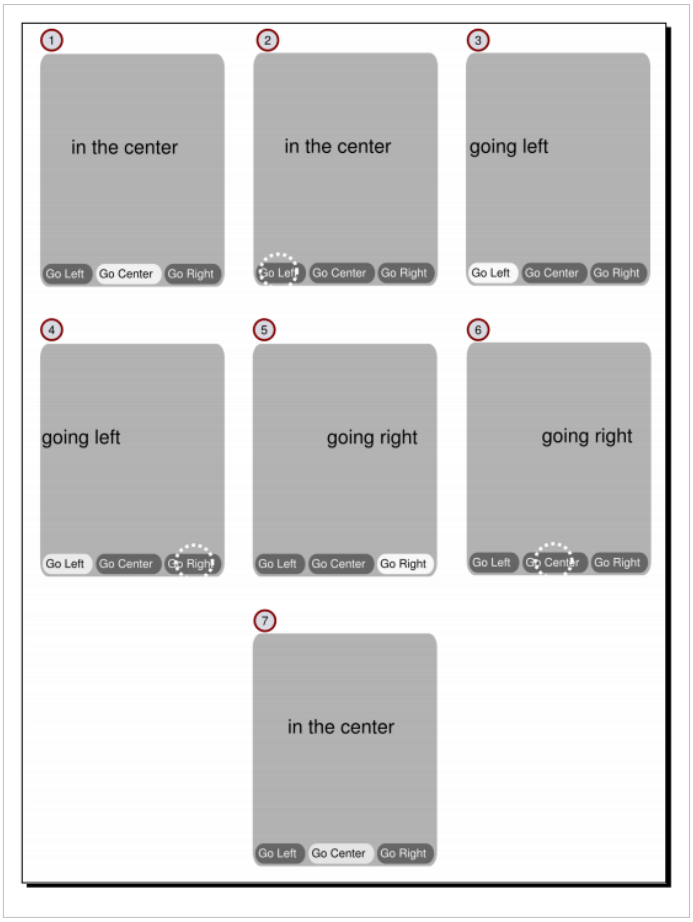
- 기본 GTK+ 애플리케이션 생성하기
- 일반 GTK+를 이용해 프로그래밍하기
- Clutter를 이용해 프로그래밍하기

시작하기 전에

Clutter로 애플리케이션을 개발하기 위해서는 제 1장, GNOME3와 SDK 설치하기에서 언급했듯이 OpenGL이 가능한 환경이 작동해야 한다. 환경이 지원되지 않으면 프로그램은 시작되지 않을 것이다. 하지만 GTK+ 애플리케이션 환경은 어떠한 하드웨어 지원도 요구하지 않는다.

기본 **GTK+** 애플리케이션 생성하기

GTK+의 기본 위젯인 라벨과 버튼의 사용부터 시작해보자. 제 2장, 무기 준비하기에서 목업(mockup)을 살펴본 적이 있는데, 이번에도 애플리케이션을 생성하기 위한 계획으로 목업을 이용하고자 한다. 그렇지만 계획을 좀 더 구체적으로 만들기 위해 목업에 기능을 추가할 것이다. 아래 목업과 같은 애플리케이션을 생성하길 원한다고 가정해보자.



목업의 각 이미지는 애플리케이션의 특정 상태를 표시한다. 한 이미지에서 다른 이미지로 따라가면 전체적인 시퀀스 흐름을 만드는데, 이를 상호작용 흐름**(interaction flow)**이라 부른다. 구체적으로 말하자면 이것은 사용자의 상호작용 직후에 애플리케이션의 반응을 표시한다. 사용자 상호작용은 그래픽 아이콘으로 표시되는데, 이는 사용자가 상호작용하는 애플리케이션 화면에서 항목의 젤 위에 그려진다. 이러한 목업에서는 클릭 상호작용만 가능한데, 이는 항목에 마우스 클릭이 실행되었음을 나타내기 위해 점으로 된 원으로 표시된다.

목업을 좀 더 가까이 살펴보자. 첫 번째 애플리케이션 상태는 이미지 번호 1로 표시된다. 이는 중앙으로 정렬된 라벨과 세 개의 버튼을 표시하는데, 가운데 버튼은 누름 상태로 되어 있다. 이미지 번호 2는 사용자가 첫 번째 버튼을 클릭함을 보여준다. 이미지 번호 3은 애플리케이션의 반응을 표시한다. 이미지가 왼쪽으로 이동하였고, 버튼 상태는 누름 상태로 변경되었음을 확인할 수 있다. 또 중간 버튼의 상태는 일반 상태로 설정되었음을 확인할 수 있다. 다음 이미지를 따라 끝까지 가면 애플리케이션에서 가능한 상호작용을 모두 갖게 된다.

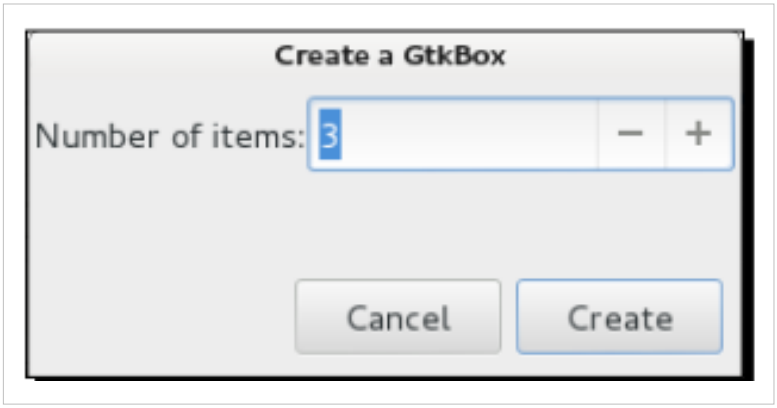
여기서 어떤 위젯을 사용해야 할 것인지 계획할 수 있다. 창의 경우 간단한 GtkWindow를 이용할 수 있겠다. 라벨은 애플리케이션 중앙에 GtkLabel을 이용할 수 있다. 버튼에 일반 GtkButton을 사용할 경우, 하나의 버튼만 눌러도 모든 버튼을 리셋해야 한다. 따라서 GtkRadioButton이라는 특수 버튼을 이용해 우리가 누르는 버튼

을 제외하고 모든 버튼을 비활성(inactive) 상태로 설정하는 기능을 구현하고자 한다. 어떤 위젯을 사용할 것인지 알고 있으니 구현을 시작할 수 있겠다.

실행하기 - 목업 구현하기

제 2장, 무기 준비하기에서 GUI 애플리케이션을 간략하게 만들어봤기 때문에 이제는 목업에 따라 라벨과 버튼을 이용하는 실험을 하기 위해 JavaScript에서 애플리케이션을 생성할 것이다.

1. Glade를 독립형 프로그램으로 실행하고 gtk-basic-widgets.uni라는 새로운 파일을 생성하여 gtk-basic-widgets라는 전용 디렉터리에 넣어라.
2. **Palette** dock에서 **Container** 섹션 안에 **Box** 객체를 찾아라. 객체를 클릭하고 오른쪽에 빈 창을 클릭하라. **Number of items** 값을 질문하면 2라고 답한다. -와 + 버튼을 사용하거나 숫자 2를 넣어도 좋다.

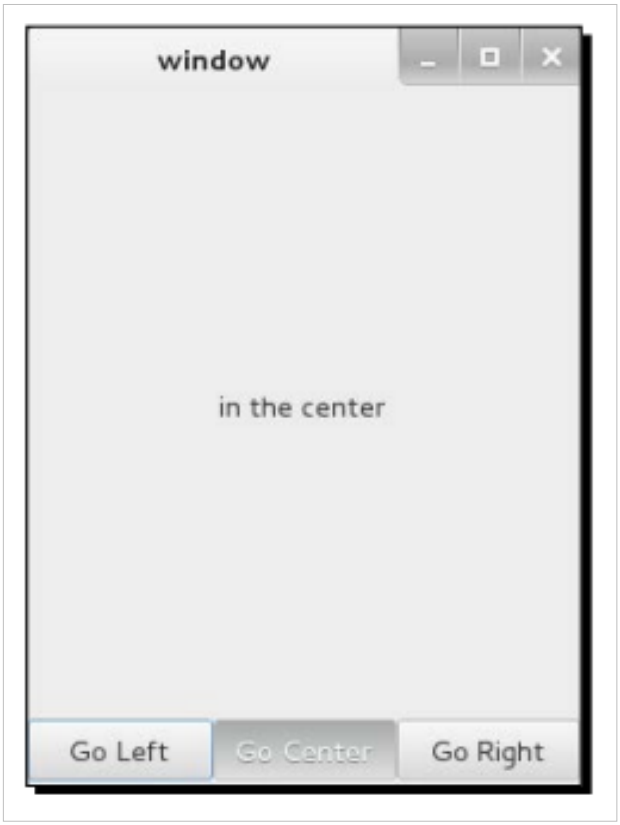


3. 이제 창의 박스가 두 개의 박스로 나뉘었음을 확인할 수 있을 것인데, 이를 상단 박스와 하단 박스라고 부르자.
4. 다시 **Container** 섹션 안에 **Box** 객체를 클릭하고 하단 박스를 클릭한다. 하단 박스는 세 개의 박스로 나뉠 것이다.
5. 두 번째 박스 그룹은 자동으로 **box2** 로 명명된다. **box2** 의 **General** 탭에서 **Orientation** 값을 확인하라. 이 값을 **Horizontal** 로 변경하라.
6. **General** 탭에서 **Homogeneous** 값을 **Yes** 로 설정하라.
7. 하단 박스가 수평으로 세 개의 박스로 나뉘었을 것이다. 이러한 박스를 왼쪽, 중앙, 오른쪽 박스라고 기억하자.
8. **Palette** dock의 **Control and Display** 섹션에서 label 을 찾아라. label 을 클릭한 후 상단 박스를 클릭하라. 이제 라벨이 상단 박스에 위치할 것이다.
9. **Widget** dock에서 **label1** 을 찾아 **Packing** 탭을 클릭하라. **Expand** 값을 클릭하여 **Yes** 로 변경하라.
10. 이제 라벨이 상단 박스로 영역을 확장한다.
11. **Palette** dock의 **Control and Display** 섹션에서 **Toggle Button** 을 찾아라. 이를 클릭한 후 왼쪽 박스를 클릭하라. **Toggle Button** 을 계속해서 클릭한 후 중앙 박스를 클릭하고, 마지막으로 오른쪽 박스를 클릭하라.
12. 중간 버튼과 오른쪽 버튼의 경우 버튼 그룹을 명시하여 선택 가능한 상태가 되도록 만들어야 한다. 이 버튼들의 **General** 탭에서 **Group** 옵션을 찾아라. 타원형 버튼(세 개의 점으로 표시)을 클릭한 후 **radiobutton1** 을 선택하라.
13. 이제 모든 버튼이 하단 박스를 채운다.
14. 중간 버튼을 클릭하고 **Active** 값을 (**General** 탭에서 찾을 수 있다) **Yes** 로 설정하라.
15. 라벨을 클릭하고 **General** 탭을 연 후 **Label** 의 값을 **in the center** 로 설정하라.
16. 왼쪽 버튼을 클릭하고 **Label on optional image** 옵션을 클릭한 후 **Label** 을 **Go Left** 로 채워라. 중간 버튼과 오른쪽 버튼에도 동일한 액션을 반복하고, 라벨을 각각 **Go Center** 와 **Go Right** 로 설정하라.
17. 각 버튼에서 **General** 탭으로 가서 **Draw indicator** 를 **No** 로, **Horizontal alignment for child** 를 **0.5** 로 설정하여 모든 버튼에 반복한다.
18. Gtk-basic-widgets.js 라는 새로운 스크립트를 생성하고 아래와 같은 코드로 채워라.

```
#!/usr/bin/env seed

Gtk= imports.gi.Gtk;
GObject = imports.gi.GObject;
Main = new GType({
    parent: GObject.Object.type,
    name: "Main",
    init: function(self) {
            this.go_left = function(object) {
    if (object.active) {
                self.label.set_label("going left");
                self.label.set_alignment(0, 0.5);
    }
            }

            this.go_center = function(object) {
    if (object.active) {
                self.label.set_label("in the center");
                self.label.set_alignment(0.5, 0.5);
    }
            }

            this.go_right = function(object) {
                if (object.active) {
    self.label.set_label("going right");
                    self.label.set_alignment(1, 0.5);
    }
            }

            var ui = new Gtk.Builder()
            this.ui = ui;
            ui.add_from_file("gtk-basic-widgets.ui");
            var window = ui.get_object("window1");
            window.resize(300, 400);
            window.show_all();
    window.signal.destroy.connect(Gtk.main_quit);

            this.label = ui.get_object("label1");

ui.get_object("radiobutton1").signal.toggled.connect(this.go_left);

ui.get_object("radiobutton2").signal.toggled.connect(this.go_center);
        ui.get_object("radiobutton2").set_active(true);

ui.get_object("radiobutton3").signal.toggled.connect(this.go_right);
    }
});
```

```
Gtk.init(Seed.argv);
var main = new Main();
Gtk.main();
```

19. 애플리케이션을 실행하라. 버튼을 눌러보고 목업이 완성되었는지 확인하라.



무슨 일이 일어났는가**?**

제 2장, 무기 준비하기에서 살펴보았듯이 창은 하나의 위젯만 취할 수 있기 때문에 위젯들(버튼과 라벨)을 창으로 직접 추가할 수는 없다. 따라서 하나 이상의 위젯을 보유할 수 있는 컨테이너 위젯이 필요하다. 바람직한 모습이 되도록 먼저 창을 두 가지 부분, 상단과 하단으로 나눈다. 이후 또 다른 컨테이너를 하단 부분에 넣는다. 이러한 컨테이너 집합을 이용해 위젯을 내부로 넣을 수 있다.

이 예제에서 우리는 GtkBox(Glade에서 **Box** 로 표시)를 컨테이너로 사용하고, 부분을 어떻게 나눌 것인지 나타내기 위해 방향을 설정한다. GtkBox는 시각적 부분을 갖고 있지 않다. 내부에 위치한 위젯들이 그곳으로 할당된 전체 영역을 덮도록 해준다. 위젯을 Box 내부에 패킹하면 위젯은 선호되는 높이에 관한 힌트를 Box로 제공하고, Box는 그 높이에 따라 영역을 계산 및 할당하여 위젯이 스스로 영역에 그려지도록 허용한다. Box는 우리가 명시한 방향에 따라 그에 추가된 위젯들을 자동으로 배열한다. 따라서 특정 좌표로 위젯을 넣지 않고 Box가 위젯을 배열하는 방식에 의존한다.

첫 번째 박스는 수직 박스로, 하단 부분은 일정한 크기로 유지하되 상단 부분은 창 크기에 따라 크기를 조정하길 원한다. 이를 구현하려면 구체적으로 상단 박스 안에 위치한 label1 위젯을 만들어 위젯이 패킹될 때 **Expand** 패킹 옵션을 **Yes** 로 설정하여 확장하면 부모의 크기가 증가할 때 라벨이 추가 공간을 제공할 것이다. 기본적으로 **Yes** 로 설정되는 **Fill** 옵션과 함께 Box는 라벨의 실제 높이를 제공하는 대신 최대 높이를 제공할 것이다.

여기서 Box는 상단과 하단 부분 안의 모든 위젯에 선호되는 크기를 계산하고 창의 크기에 따라 이용할 수 있는 나머지 크기를 알아내는 일을 실행한다. 이후 나머지 크기를 라벨의 원래 크기로 추가하여 상단 부분 안에 있는 label1 위젯의 크기를 최대화한다.

하단 부분을 직접 나눌 수는 없는 것은 Box 객체가 하나의 방향만 가질 수 있기 때문이다. 직접 나누기 위해서는 하단 부분 안에 새로운 수평적 Box 위젯을 추가해야 한다. 이러면 상황은 약간 달라진다. 여기서 Box 방향은 수평적이고 세 개의 부분으로 균등하게 나누길 원하므로 **Homogeneous** 옵션을 활성화할 필요가 있다.

토글 버튼에서 우리는 radiobutton1을 그룹 리더로 사용한다. 이것을 이용할 경우 하나의 버튼을 누르면 나머지 버튼들은 비활성(inactive)가 된다. **Draw indicator** 옵션을 비활성화하고 버튼 내 모든 라벨을 중앙으로 정렬함으로써 일반 버튼처럼 보이도록 만들기도 해야 한다.

이 예제에서 우리는 상속구조에 다수의 위젯을 넣었는데, 이는 어떤 위젯들에겐 부모와 자식이 있다는 뜻이다. 가령 창에는 자체의 자식들과 창에 부착된 모든 위젯의 자식들이 포함된다. 수직적 Box 위젯은 라벨의 자식들, 수평적 Box 위젯과 그의 자식들을 갖게 된다. 수평적 Box 위젯은 모든 버튼을 그 자식으로 갖는 식이다. 상속구조를 이해하는 것은 애플리케이션의 구조체를 이해하는 데에 매우 중요하다. 상속구조는 Glade에서 **Inspector** dock에서 확인할 수 있다.

```
Gtk.init(Seed.argv);
    var main = new Main();
    Gtk.main();
```

여기에 GTK+의 초기화 코드가 있다. Gtk.init는 그래픽 환경을 준비하여 애플리케이션이 GUI 애플리케이션으로서 실행되도록 준비한다. Seed.argv는 애플리케이션으로 전달되는 인자의 리스트를 제공한다. 이번 예제에서는 빈 리스트가 되겠다. 그리고 Main 클래스를 초기화한다. 이후 Gtk.main을 호출함으로써 GTK+ 메인 루프를 입력한다.

이 코드 부분에서 Gtk.Builder 객체를 이용해 gtk-basic-widgets.ui 파일을 로딩한다.

```
this.ui = new Gtk.Builder()
this.ui.add_from_file("gtk-basic-widgets.ui");
```

경로가 올바른지 확인하라.

창의 이름, 즉 window1을 검색하여 창의 참조를 얻는다.

```
    var window = this.ui.get_object("window1");
```

이름을 변경하면 이 코드에서도 마찬가지로 변경해야 한다.

이름만 올바르다면 Glade에서 사용하는 위젯은 무엇이든 get_object 함수를 이용해 찾을 수 있다.

이제 창의 크기를 조정하고 창의 내용을 모두 화면에 표시한다.

```
window.resize(300, 400);
    window.show_all();
```

이 부분은 창이 닫힐 때마다 애플리케이션을 종료할 것이다.

```
        window.signal.destroy.connect(Gtk.main_quit);
```

이 코드가 없이는 창이 닫혀도 애플리케이션이 계속 실행될 것이다. 애플리케이션이 아직도 실행 중인지 확인하는 방법으로, terminal에 ps 명령을 실행시키는 방법이 있다.

아래 코드에서는 모든 버튼의 토글 시그널(toggled signal)을 각각의 핸들러로 연결한다. 토글 시그널은 상태가 활성화에서 비활성 상태로 혹은 그 반대로 변경될 때마다 발생한다.

```
this.label = ui.get_object("label1");

ui.get_object("radiobutton1").signal.toggled.connect(this.go_left);
```

```
ui.get_object("radiobutton2").signal.toggled.connect(this.go_center);
ui.get_object("radiobutton2").set_active(true);


ui.get_object("radiobutton3").signal.toggled.connect(this.go_right);
```

radiobutton1 버튼이 토글되면 시그널 핸들러 go_left 함수가 호출된다. radiobutton2의 경우 go_center가 호출되고, radiobutton3이 토글되면 go_right가 호출된다. radiobutton2의 경우 활성화(active)로 초기화해야만 목업의 초기 상태와 일치한다.

go_left 함수에서 라벨을 **going left** 로 설정하고, set_alignment 함수의 첫 번째 변수를 0으로 설정함으로써 라벨을 왼쪽으로 정렬한다. 두 번째 변수는 수직 정렬용이다.

```
this.go_left = function(object) {
        if (object.active) {
            self.label.set_label("going left");
                self.label.set_alignment(0, 0.5);
        }
    }
```

이는 항상 0.5로 설정된다. 정렬값 범위는 0(왼쪽)부터 1(오른쪽)까지다. 수직 정렬값 또한 0(상단)부터 1(하단)까지다. Glade에서 모든 버튼에 **Horizontal alignment for child** 를 0.5로 설정하면 버튼 안의 라벨이 중앙 정렬로 설정된다는 뜻임을 기억하라.

우리는 버튼이 활성화되었을 때만 이러한 설정(settings)을 설정하길 원한다. 따라서 if (object.active)를 이용해 함수를 보호해야만 라벨과 정렬 설정이 두 번 호출되는 것을 피할 수 있다. 이 예제에서는 가드(guard)를 갖고 있지 않더라도 그다지 차이가 없다. 하지만 실제 라이브 애플리케이션에서는 중복 호출을 피하기 위해 종종 가드가 필요한데, 중복 호출은 혼동을 야기하기도 하고 핸들러 내에 복잡한 계산이 있어 성능을 저하시키기도 하기 때문이다.

go_center와 go_right 함수에도 비슷한 일이 발생한다. 하지만 이 함수에서는 0.5 를 인자로 하여 중앙 정렬로, 1을 인자로 하여 오른쪽 정렬로 설정한다.

팝 퀴즈

Q1. 수평 정렬 값이 0.3일 경우 위젯의 시각적인 위치는 어디인가?

1. 중앙
2. 약간 오른쪽
3. 약간 왼쪽

Q2. 수직 정렬 값이 0.3일 경우 위젯의 시각적인 위치는 어디인가?

1. 중앙
2. 약간 오른쪽
3. 약간 왼쪽

시도해보기 - Vala 버전 생성하기

제 2장 무기 준비하기에서 사용한 예제를 이용해 Vala 버전으로 간단한 코드를 이식해볼 수 있겠다. 이러한 이식은 간단하고 수월해야 한다.

실행하기 - 아이콘을 버튼에 추가하기

이제 아이콘을 버튼으로 추가하길 원한다고 가정해보자. 아이콘은 주로 버튼에 의도된 기능을 설명하기 위해 버튼으로 추가된다. 버튼의 라벨이 짧거나 모호할 경우 더 중요해질 것이다. 아이콘을 버튼으로 추가하기 위해서는 아래 단계를 따라 해보자.

1. gtk-basic-widgets.ui에 있을 때 왼쪽 버튼을 클릭하라.

2. **General** 탭을 찾고, **Label with option image** 옵션 아래의 **image widget** 옵션을 찾아라.
3. 타원형 버튼을 클릭하면 대화상자가 나타날 것이다.
4. **New** 버튼을 클릭하면 이미지 위젯이 생성될 것이다.
5. 이제 위젯 리스트에서 위젯을 찾아라. 처음에 image1로 명명될 것이다.
6. 이미지를 클릭하고 **General** 탭을 찾아 **Edit image** 아래의 **Stock ID** 를 찾아라. 이를 클릭하고 **Left** 아이콘을 찾아라. 아이콘명은 gtk-justify-left다.
7. 위의 과정을 반복하되 중간 버튼과 오른쪽 버튼에 **Center (gtk-justify-center)** 와 **Right(gtk-justify-right)** 아이콘을 사용하라.
8. 코드에서 생성자 내에 아래의 행을 추가하라.

```
var s = Gtk.Settings.get_default();
s.gtk_button_images = true;
```

9. 이를 실행하라.

무슨 일이 일어났는가**?**

애플리케이션의 위젯 상속구조 밖에서 이미지가 생성되었다. 즉, 부모 위젯이나 자식 위젯이 없다는 의미다. 그저 코드 "어딘가에" 생성되었을 뿐이다. 이미지는 버튼으로 묶음으로써 사용한다. 이미지는 기본적으로 표시되지 않을 것이다.

```
var s = Gtk.Settings.get_default();
    s.gtk_button_images = true;
```

위의 코드를 이용하면 GTK+가 이미지를 표시한다.

시도해보기 - 아이콘 위치조정(placement)

자, 이제 **Widgets** dock의 **General** 탭에서 이용 가능한 프로퍼티를 살펴보라. 아이콘의 위치를 조정하는 데에 사용할 수 있는 **Image position** 프로퍼티가 보일 것이다. 이 프로퍼티를 이용하면 오른쪽 정렬 아이콘을 버튼의 오른쪽에, 중앙 정렬 아이콘을 버튼의 중앙에 위치하도록 설정할 수 있다. 프로퍼티의 스위치를 활성화시키면 될 일이다!

GtkBuilder 없이 코드 이식하기

지금까지는 Glade를 이용해 UI를 디자인하고 런타임 시 디자인을 로딩하기 위해서는 GtkBuilder를 사용해왔다. 본 서적의 나머지 부분에서는 Glade를 사용할 것이지만 현재로선 Glade를 사용할 경우 분명하지 않은 GTK+의 기본 내용을 이해하기 위해 저수준에서 GTK+ 프로그래밍을 실행하도록 하겠다. 프로그램이 갈수록 커지면서 언젠가 성능 문제, 레이아웃 문제 등에 직면하게 될 것이기 때문에 이러한 지식은 더욱 더 중요해진다.

실행하기 - raw GTK+를 이용해 프로그래밍하기

목업과 상호작용 흐름을 구현하기 위해 raw GTK+ 프로그래밍을 사용하는 JavaScript 코드를 실행할 것이다.

1. gtk-basic-widgets-sans-glade.js 라는 JavaScript 파일을 그 이름에서 .js 확장자만 제외한 디렉터리에서 생성하라.
2. 스크립트에 아래 코드를 사용하라.

```
#!/usr/bin/env seed

Gtk = imports.gi.Gtk;
GObject = imports.gi.GObject;

Main = new GType({
    parent: GObject.Object.type,
    name: "Main",
    init: function(self) {
        this.go_left = function(object) {
            if (object.active) {
                    self.label.set_label("going left");
                    self.label.set_alignment(0, 0.5);
            }
                }

                this.go_center = function(object) {
            if (object.active) {
                    self.label.set_label("in the center");
                    self.label.set_alignment(0.5, 0.5);
            }
                }

                this.go_right = function(object) {
                    if (object.active) {
            self.label.set_label("going right");
                        self.label.set_alignment(1, 0.5);
            }
                }
```

```
        var window = new Gtk.Window();
        window.signal.destroy.connect(Gtk.main_quit);

        var topBottomBox = new Gtk.Box();
        topBottomBox.orientation = Gtk.Orientation.VERTICAL;
        topBottomBox.set_homogeneous(false);

        window.add(topBottomBox);

        var label = new Gtk.Label();
        this.label = label;
        label.set_text("in the center");
        topBottomBox.pack_start(label, true, true, 0);

        var buttonBox = new Gtk.Box();
        buttonBox.set_homogeneous(true);

        topBottomBox.pack_start(buttonBox, false, false, 0);

        var leftButton = new Gtk.RadioButton.with_label(null, "Go
left");
        var centerButton = new
Gtk.RadioButton.with_label_from_widget(leftButton, "Go center");
        var rightButton = new
Gtk.RadioButton.with_label_from_widget(leftButton, "Go right");

        leftButton.signal.clicked.connect(this.go_left);
        centerButton.signal.clicked.connect(this.go_center);
        rightButton.signal.clicked.connect(this.go_right);

        leftButton.draw_indicator = centerButton.draw_indicator =
rightButton.draw_indicator = false;

        centerButton.active = true;

        leftButton.xalign = 0.5;
        centerButton.xalign = 0.5;
        rightButton.xalign = 0.5;
        buttonBox.pack_start(leftButton, false, true, 0);
        buttonBox.pack_start(centerButton, false, true, 0);
        buttonBox.pack_start(rightButton, false, true, 0);

        window.show_all();
        window.resize(300, 400);
    }
});
```

```
Gtk.init(Seed.argv);
var main = new Main();

Gtk.main();
```

3. 위를 실행하라. Glade와 GtkBuilder를 사용하는 버전과 시각적으로 동일할 것이다.

무슨 일이 일어났는가?

앞의 실험과 비슷한 코드 부분에 대해서는 논하지 않고 수동 위젯의 선언을 좀 더 자세히 살펴보겠다.

먼저 GtkWindow를 생성하고 destroy 시그널을 연결하여 창이 닫힐 때마다 애플리케이션이 바람직하게 종료되도록 할 것이다.

```
var window = new Gtk.Window();
        window.signal.destroy.connect(Gtk.main_quit);
```

이 부분에서 창을 상단 부분과 하단 부분으로 나누는 박스를 준비한다. 이 때 박스는 수직적이고 균일하지 않은(non-homogeneous) 박스임을 명시적으로 나타낸다. 이후 자식이 하나인 컨테이너 위젯, 즉 창으로 박스를 추가한다.

```
var topBottomBox = new Gtk.Box();
topBottomBox.orientation = Gtk.Orientation.VERTICAL;
topBottomBox.set_homogeneous(false);
    window.add(topBottomBox);
```

다음으로 라벨을 준비하여 상단 박스로 패킹한다. Glade에서는 박스에 위치시킬 항목의 개수를 명시하는 반면 수동 GTK+ 프로그래밍에서는 항목의 개수를 미리 선언해야 하는 가능성 없이 항목을 하나씩 박스로 패킹하면 된다. Glade는 항목을 놓을 수 있는 플레이스홀더를 준비하기 위한 용도로 숫자를 필요로 한다.

```
var label = new Gtk.Label();
this.label = label;
label.set_text("in the center");
topBottomBox.pack_start(label, true, true, 0);
```

이 코드에서는 true로 설정된 Fill과 Expand 프로퍼티의 값을 이용해 label 변수를 패킹한다. 0 값은 패킹 중에 어떠한 패딩도 추가해선 안 됨을 의미한다.

이후 버튼을 위해 박스를 준비시킨다. 방향은 기본적으로 수평으로 설정되어 있기 때문에 별도로 손대지 않고 homogeneous 프로퍼티만 설정한다. 그리고 Fille과 Expand 값을 설정하지 않고 패킹하면 된다.

```
var buttonBox = new Gtk.Box();
buttonBox.set_homogeneous(true);
    topBottomBox.pack_start(buttonBox, false, false, 0);
```

여기서 첫 번째 버튼을 생성한다.

```
        var leftButton = new Gtk.RadioButton.with_label(null, "Go
left");
```

다음으로 leftButton을 그룹 리더로 하여 두 개의 버튼이 추가된다.

```
var centerButton = new
Gtk.RadioButton.with_label_from_widget(leftButton, "Go center");
    var rightButton = new
Gtk.RadioButton.with_label_from_widget(leftButton, "Go right");
```

이 부분은 아래 버튼의 클릭된 시그널을 연결한다.

```
leftButton.signal.clicked.connect(this.go_left);
centerButton.signal.clicked.connect(this.go_center);
rightButton.signal.clicked.connect(this.go_right);
```

여기서 draw_indicator 프로퍼티를 false로 설정하면 라디오 버튼의 모양이 일반 버튼과 같아질 것이다.

```
    leftButton.draw_indicator = centerButton.draw_indicator =
rightButton.draw_indicator = false;
```

아래는 목업의 첫 번째 상태에 표시된 바와 같이 중간 버튼을 활성화하기 위함이다.

```
    centerButton.active = true;
```

아래 부분은 버튼 안에 라벨의 정렬을 설정한다. 0.5 값은 라벨 모두 중앙으로 정렬됨을 의미한다.

```
leftButton.xalign = 0.5;
centerButton.xalign = 0.5;
    rightButton.xalign = 0.5;
```

이후 Fill 프로퍼티만 true로 설정되도록 하여 버튼을 패킹한다.

나머지 코드 부분은 GtkBuilder가 가능한(GtkBuilder-enabled) 코드와 비슷하다. 여기서는 수동 GTK+ 프로그래밍이 GTK+ API에 관해 더 많은 정보를 제공할 수 있음을 확인할 수 있다.

```
buttonBox.pack_start(leftButton, false, true, 0);
buttonBox.pack_start(centerButton, false, true, 0);
    buttonBox.pack_start(rightButton, false, true, 0);
```

프로젝트의 .ui 파일이 커질수록 시동 시 파일을 로딩하는 데에 시간이 더 소요된다. 이런 경우 수동으로 패킹하는 방도를 고려하는 것도 좋은데, 특히 .ui 파일이 충분히 안정적이어서 개발 중에 많이 변경되지 않을 경우 그렇다. 반면 .ui 파일이 계속 변경될 경우 Glade와 GtkBuilder의 사용을 고려하는 것이 나은데, 수동 GTK+는 종종 머리를 어지럽게 만들기도 하기 때문이다. 불행히도 API 자체가 그다지 직관적이지 못하고 쉽게 오용되기 때문이다.

하지만 사용자 인터페이스를 깔끔하고 잘 구조화된 채로 유지하는 것이 좋은데, 그래야 어떤 방법이든 선택할 수 있기 때문이다. 목업을 이용해 어떤 위젯을 이용할 것인지 미리 계획하는 것도 좋은 실습이 되겠다.

Clutter를 이용한 GUI 프로그래밍

Clutter는 풍부한 애니메이션과 효과적인 기능을 제공한다는 이유로 주로 좀 더 강력한 GUI 애플리케이션을 생성하는 데에 사용되며, 종종 OpenGL을 이용한 렌더링에 사용되기도 한다. 아직은 OpenGL 특정적인 프로그래밍은 모두 개발자들에게 숨긴다. GTK+와 달리 Clutter는 scene graph 기반의 캔버스로, 원하는 장소로 무엇이든 놓을 수 있도록 해준다. 스테이지(stage) 상의 모든 객체는 2D 평면이지만 스테이지 자체는 3D 공간이다.

실행하기 - Clutter를 이용해 목업 구현하기

Clutter를 이용해 목업과 그 상호작용 흐름을 구현함으로써 Clutter를 소개하겠다.

1. clutter-basic-vala라고 불리는 디렉터리에 clutter-basic.js라고 불리는 새로운 JavaScript를 생성하라.
2. 아래 코드로 스크립트를 채워라.

```
#!/usr/bin/env seed
```

```
Clutter = imports.gi.Clutter;
Pango = imports.gi.Pango;
GObject = imports.gi.GObject;

Main = new GType({
    parent: GObject.Object.type,
    name: "Main",
    init: function(self) {
        var stageColor = new Clutter.Color();
        stageColor.from_string("#b0b0b0");

        var labelColor = new Clutter.Color();
        labelColor.from_string("#000000");

        var buttonColor = new Clutter.Color();
        buttonColor.from_string("#505050");

        var buttonPressedColor = new Clutter.Color();
        buttonPressedColor.from_string("#a0a0a0");

        var buttonTextColor = new Clutter.Color();
        buttonTextColor.from_string("#000000");

        var buttonLeft = new Clutter.Rectangle();
        buttonLeft.width = 100;
        buttonLeft.height = 40;
        buttonLeft.x = 0;
        buttonLeft.y = 360;
        buttonLeft.color = buttonColor;
        buttonLeft.set_border_color(stageColor);
        buttonLeft.set_border_width(1);

        var buttonCenter = new Clutter.Rectangle();
        buttonCenter.width = 100;
        buttonCenter.height = 40;
        buttonCenter.x = 100;
        buttonCenter.y = 360;
        buttonCenter.color = buttonColor;
        buttonCenter.set_border_color(stageColor);
        buttonCenter.set_border_width(1);

        var buttonRight = new Clutter.Rectangle();
        buttonRight.width = 100;
        buttonRight.height = 40;
        buttonRight.x = 200;
        buttonRight.y = 360;
        buttonRight.color = buttonColor;
```

```
        buttonRight.set_border_color(stageColor);
        buttonRight.set_border_width(1);

        var s = Clutter.Stage.get_default();
        s.color = stageColor;
        this.s = s;
        s.width = 300;
        s.height = 400;
        var fd = Pango.FontDescription.from_string("Sans 16");
        var label = new Clutter.Text();
        label.set_font_description(fd);
        label.set_text("in the center");
        label.color = labelColor;
        label.x = (s.width - label.width)/2;
        label.y = 100;

        var buttonFd = Pango.FontDescription.from_string("Sans 12");

        var buttonLeftText = new Clutter.Text();
        buttonLeftText.set_font_description(buttonFd);
        buttonLeftText.set_text("Go Left");
        buttonLeftText.color = buttonTextColor;
        buttonLeftText.x = (buttonLeft.width - buttonLeftText.width)
/2;
        buttonLeftText.y = (buttonLeft.height - buttonLeftText.height)
/2 + buttonLeft.y;

        var buttonCenterText = new Clutter.Text();
        buttonCenterText.set_font_description(buttonFd);
        buttonCenterText.set_text("Go Center");
        buttonCenterText.color = buttonTextColor;
        buttonCenterText.x = (buttonCenter.width -
buttonCenterText.width) /2 + buttonLeft.width;
        buttonCenterText.y = (buttonCenter.height -
buttonCenterText.height) /2 + buttonCenter.y;

        var buttonRightText = new Clutter.Text();
        buttonRightText.set_font_description(buttonFd);
        buttonRightText.set_text("Go Right");
        buttonRightText.color = buttonTextColor;
        buttonRightText.x = (buttonRight.width - buttonRightText.width)
 /2 + buttonLeft.width + buttonCenter.width;
        buttonRightText.y = (buttonRight.height -
buttonRightText.height) /2 + buttonRight.y;

        buttonLeft.set_reactive(true);
        buttonLeft.signal.button_press_event.connect(function(self) {
```

```
            buttonLeft.color = buttonPressedColor;
            buttonRight.color = buttonCenter.color = buttonColor;
            label.save_easing_state();
            label.set_text("going left");
            label.set_x(0);
            label.restore_easing_state();
            return true;
        });

        buttonCenter.set_reactive(true);
        buttonCenter.signal.button_press_event.connect(function(self)
{
            buttonCenter.color = buttonPressedColor;
            buttonRight.color = buttonLeft.color = buttonColor;
            label.save_easing_state();
            label.set_text("in the center");
            label.set_x((s.width - label.width)/2);
            label.restore_easing_state();

            return true;
        });

        buttonRight.set_reactive(true);
        buttonRight.signal.button_press_event.connect(function(self) {
            buttonRight.color = buttonPressedColor;
            buttonLeft.color = buttonCenter.color = buttonColor;
            label.save_easing_state();
            label.set_text("going right");
            label.set_x(s.width - label.width);
            label.restore_easing_state();

            return true;
        });

        buttonCenter.color = buttonPressedColor;

        s.add_actor(buttonLeft);
        s.add_actor(buttonRight);
        s.add_actor(buttonCenter);
        s.add_actor(buttonLeftText);
        s.add_actor(buttonRightText);
        s.add_actor(buttonCenterText);

        s.add_actor(label);
        s.show_all();
    }
});
```

```
Clutter.init(Seed.argv);
var main = new Main();

Clutter.main();
```

3. JavaScript 코드 대신 Vala를 원한다면 Anjuta에서 프로젝트를 생성할 때 non-GUI 애플리케이션 설정과 유사한 설정을 이용해 새로운 Vala 프로젝트를 생성해보자. clutter-basic-vala라는 새 프로젝트를 생성해보자.
4. src/Makefile.am 파일을 편집하고 아래 부분을 찾아라.

```
clutter_basic_vala_VALAFLAGS = \
    --pkg gtk+-3.0
```

   두 행을 아래의 내용으로 편집하라.

```
clutter_basic_vala_VALAFLAGS = \
    --pkg clutter-1.0
```

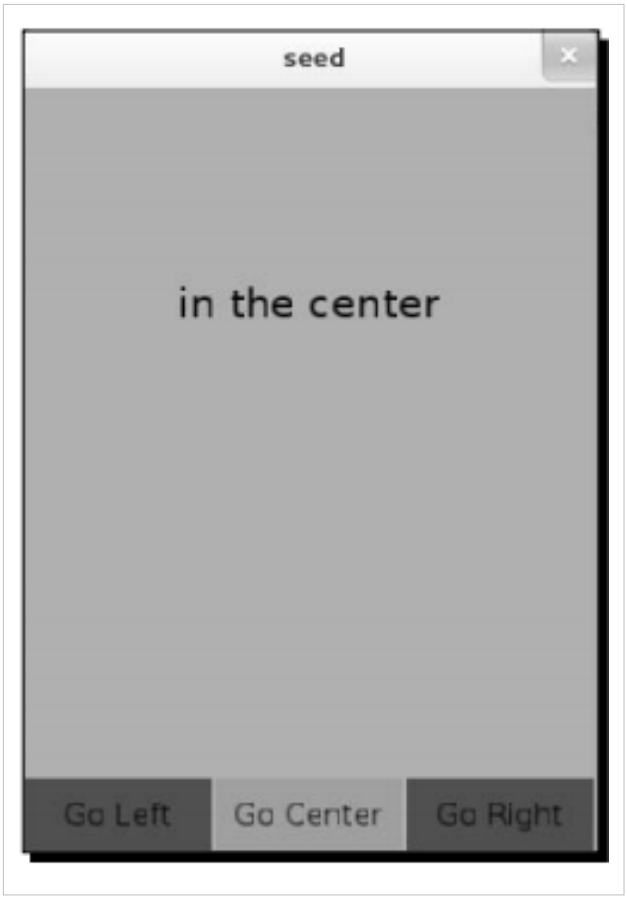
5. configure.ac 파일을 편집하고 아래 행을 찾아라.

```
PKG_CHECK_MODULES(CLUTTER_BASIC_VALA, [gtk+-3.0])
```

   전체 행을 아래 행으로 대체하라.

```
PKG_CHECK_MODULES(CLUTTER_BASIC_VALA, [clutter-1.0])
```

6. 본문에서 코드를 반복해서 싣는 것을 피하기 위해 JavaScript 코드를 Vala로 이식해보자. 앞에서 두 개의 프로그래밍 언어를 소개한 바가 있기 때문에 꽤 수월한 작업이다.
7. 프로그램을 실행하면 아래 스크린샷과 같은 모습이 될 것이다.



무슨 일이 일어났는가**?**

코드의 양이 많다는 것은 단숨에 알아볼 수 있다. 이는 Clutter가 GTK+보다 저수준이기 때문이다. 이것은 매우 기본적인 위젯만 갖고 있기 때문에 처음부터 고유의 위젯을 생성해야 한다. Clutter가 제공하는 함축적 애니메이션 때문에 상호작용은 좀 더 유동적이고 만족스럽다.

기술적 측면에서 보면 객체가 스테이지 어디든 위치할 수 있음을 확인할 수 있다. Clutter에서 객체는 행위자 **(actors)** 라고 부른다. 애니메이션의 역할이나 효과를 행위자로 적용할 수 있고 스테이지 주위로 이동시킬 수 있다.

JavaScript와 비교할 때 Vala로 Clutter를 코딩할 때는 시작 시 약간의 준비가 필요하다. configure.ac와 Makefile.am 파일을 수정함으로써 빌드 시스템을 조정할 필요가 있다.

이러한 조정을 거치면 빌드 시스템은 Vala에서 컴파일할 때나 생성된 C 코드 파일을 만들 때 Clutter 라이브러리를 익히게 된다. 우리는 Makefile.am과 configure.ac에서 gtk+-3.0 설정을 제거함으로써 GTK+ 라이브러리를 제거하고 이를 clutter-1.0으로 대체하는 일을 수행한 것이다. 그 이유는 이번 예제에서 GTK+가 전혀 필요하지 않기 때문이다. JavaScript를 이용하면 자동으로 작동할 것이기 때문에 우리는 어떤 일도 수행할 필요가 없으므로 훨씬 더 수월하다.

Clutter가 어떻게 작동하는지 이해하도록 코드를 좀 더 깊이 살펴보자.

메인 코드는 꽤 간단하고 GTK+의 메인 코드와 유사하다. init와 메인 루프 함수가 있다.

```
Clutter.init(Seed.argv);
var main = new Main();
Clutter.main();
```

먼저 애플리케이션에서 사용되는 색상 몇 가지를 정의한다. 이는 16진 RGB 색상 포맷을 파싱한 후 변수로 저장함을 확인할 수 있다.

```
    var stageColor = new Clutter.Color();
    stageColor.from_string("#b0b0b0");
```

이것을 우리의 버튼 위젯이라고 상상해보자. 코드를 간소화하기 위해 크기와 위치를 하드코딩한 후 색상을 앞서 설정한 색상 중 하나로 설정한다.

실제 애플리케이션에서 크기와 위치는 계산 또는 사전 정의된 설정으로부터 비롯되어야 한다.

```
    var buttonLeft = new Clutter.Rectangle();
    buttonLeft.width = 100;
    buttonLeft.height = 40;
    buttonLeft.x = 0;
    buttonLeft.y = 360;
    buttonLeft.color = buttonColor;
    buttonLeft.set_border_color(stageColor);
    buttonLeft.set_border_width(1);
```

이 부분은 모든 위젯을 위치시킬 스테이지를 정의한다. 스테이지는 앞서 소개한 GtkWindow 위젯에 비할 수 있다.

```
    var s = Clutter.Stage.get_default();
    s.color = stageColor;
    this.s = s;
    s.width = 300;
    s.height = 400;
```

이 코드 부분은 글꼴 설명을 정의한다. 문자열로부터 설명을 취하여 후에 라벨이 사용하게 될 글꼴의 논리적 표현을 리턴한다.

```
    var fd = Pango.FontDescription.from_string("Sans 16");
```

그리고 라벨의 텍스트, 글꼴, 색상, 위치를 정의한다.

```
    var label = new Clutter.Text();
    label.set_font_description(fd);
    label.set_text("in the center");
    label.color = labelColor;
    label.x = (s.width - label.width)/2;
    label.y = 100;
```

여기서는 버튼의 텍스트를 정의한다. 모습이 개선되었음을 확인할 수 있는데, 즉 텍스트의 위치를 더 이상 하드코딩하지 않는다는 점이 확인된다.

```
var buttonFd = Pango.FontDescription.from_string("Sans 12");
var buttonLeftText = new Clutter.Text();
buttonLeftText.set_font_description(buttonFd);
buttonLeftText.set_text("Go Left");
buttonLeftText.color = buttonTextColor;
buttonLeftText.x = (buttonLeft.width - buttonLeftText.width) /2;
buttonLeftText.y = (buttonLeft.height - buttonLeftText.height) /2 +
buttonLeft.y;
```

다음으로 버튼을 반응(reactive)하도록 설정한다. 이 코드가 없다면 버튼은 들어오는 이벤트에 반응할 수 없을 것이다.

```
buttonLeft.set_reactive(true);
```

아래는 button-press-event 이벤트의 핸들러를 정의한다. 핸들러에서는 누름 버튼을 설정하고 눌러진 느낌을 표현하기 위해 다른 버튼의 색상을 리셋한다. 라벨의 프로퍼티를 변경하고 후에 이를 복구하기 전에 easing 상태를 저장함으로써 함축적 애니메이션을 요청한다. 함축적 애니메이션이란 코드에서 우리가 특정 타입의 애니메이션을 구체적으로 요청하지 않음을 의미한다. 대신 우리가 행위자에서 프로퍼티를 변경할 때마다 Clutter가 애니메이션을 실행하도록 의존한다.

```
buttonLeft.signal.button_press_event.connect(function(self) {
    buttonLeft.color = buttonPressedColor;
    buttonRight.color = buttonCenter.color = buttonColor;
    label.save_easing_state();
    label.set_text("going left");
    label.set_x(0);
    label.restore_easing_state();

    return true;
});
```

값을 변경하자마자 Clutter는 값을 목표값(target value)으로 취급하고, 타이머를 준비하여 타이머의 각 프레임마다 위젯을 향한 프로퍼티의 현재 값을 증가(또는 감소)시킬 것이다. 이 모든 것은 배경에서 자동으로 발생한다.

actor 라벨에서 Clutter는 애니메이션에 두 가지, 즉 텍스트와 텍스트의 위치에 변경내용을 적용한다. 위치는 텍스트의 새로운 너비를 필요로 하는데 이는 텍스트 값을 설정한 다음에 이용할 수 있으므로 이번 예제에서는 위치를 변경하기 전에 텍스트를 설정해야 함을 명심하라. 순서를 바꾸면 라벨의 위치는 올바르지 않은 값이 될 것이다.

한 가지 관찰해야 할 사항이 있는데, 시그널에 대해 반응할 때 무언가 조치를 취할 때마다 함수가 true를 리턴한다는 점이다. 혹시 시그널을 건너뛰거나 시그널에 대한 조치를 거부할 경우에는 false를 리턴해야만 큐에 있는 또 다른 이벤트 핸들러가 시그널을 처리할 것이다.

```
s.add_actor(buttonLeft);
s.add_actor(buttonRight);
s.add_actor(buttonCenter);
s.add_actor(buttonLeftText);
s.add_actor(buttonRightText);
s.add_actor(buttonCenterText);
s.add_actor(label);
```

여기서는 스테이지에 모든 행위자를 추가하여 아래와 같이 표시되도록 한다.

```
s.show_all();
```

마지막으로 모든 행위자와 스테이지를 함께 표시한다.

시도해보기 - 애니메이션 갖고 놀기

버튼이나 라벨에는 수많은 변화를 적용할 수 있다. 예를 들어, 버튼을 누르면 버튼의 크기를 증가시킬 수 있고 버튼 누름을 해제하면 버튼이 다시 본래 크기로 작아지도록 만들 수 있다. 그리고 이 모든 것이 애니메이션으로 만들어질 것이다. button-release-event 이벤트를 처리하도록 더 많은 클로저를 추가하여 애니메이션으로 만들고자 하는 행위자의 본래 값을 복구할 수도 있다.

요약

이번 장에서는 GTK+와 Clutter를 이용한 GUI 애플리케이션의 생성과 관련해 학습하였다. 각 툴킷에 대한 시스템 요구사항에는 차이가 있음을 이해하였고, 이러한 툴킷을 이용해 생성할 수 있는 애플리케이션의 유형도 학습하였다.

GTK+를 이용하면 사용할 준비가 된 위젯이 제공된다는 사실도 학습하였다. 그 중에서도 Button, Box, Window, Label의 사용법을 익혔다. 위젯의 정렬을 설정하고, 그들의 프로퍼티를 설정함으로써 관리하는 방법도 학습하였다. 이벤트에 반응하고 유용한 일을 실행할 핸들러로 이벤트를 연결하는 방법도 알게 되었다.

Clutter를 이용하면 만족스러운 시각적 애니메이션이 있는 애플리케이션을 생성할 수 있음을 배웠다. 하지만 여기에도 몇 가지 한계가 있다. 그 중 하나로, 애플리케이션을 만들 때 용이하게 사용하기엔 너무 기본적인 기반만 제공한다는 점을 들 수 있다.

그 외에도 목업과 상호작용 흐름을 이용하면 GUI 애플리케이션의 개발이 훨씬 수월해질 수 있음을 배웠다. 또 애플리케이션에 어떤 위젯을 사용할 것인지 미리 계획할 수 있음을 알게 되었다.

다음 장에서는 자신만의 위젯을 생성함으로써 GTK+를 확장하는 법을 학습할 것이다.

Notes

GNOME3ApplicationDevelopmentBeginnersGuide:Chap 06

제 6 장 위젯 생성하기

위젯 생성하기

앞 장에서 살펴보았듯이 *GTK+*는 표준 위젯을 제공한다. 어떤 경우에는 애플리케이션에서 *GTK+* 표준 위젯만 이용하더라도 살아남을 수 있다. 하지만 애플리케이션 디자인과 요구사항이 더 복잡해질수록 자신만의 위젯을 구현해야 하는 일은 피할 수 없다. 특히 비슷한 디자인과 요구사항을 공유하는 여러 개의 프로젝트를 실행할 때 그러하다. 자신만의 위젯을 구현하지 않으면 결국 코드를 복사할 수 밖에 없어 결국 관리 측면에서 끔찍한 일이 될 것이다.

자신만의 위젯을 구현한다는 것은 본래 위젯에서 이용할 수 없는 기능을 추가하거나 제거하는 등 위젯의 맞춤설정을 의미하기도 한다. 이번 장에서는 처음부터 GTK+ 위젯을 생성하는 방법과 확장하는 방법까지 살펴볼 것이다. 또 Cairo canvas API를 이용하여 내부적으로 위젯을 그리는 방법을 논하고자 한다.

이번 장에서 구체적으로 다룰 주제는 다음과 같다.

- 위젯 오버라이드하기
- 위젯에 새로운 기능 추가하기
- 커스텀 위젯 구현하기
- 컴파일된 라이브러리에서 위젯 관리하기

시작해보자.

시작하기 전에

이번 장에서는 Glade를 사용하는 대신 raw GTK+ 프로그래밍을 사용할 것이다. 따라서 Vala 프로젝트는 제 *3* 장, 프로그래밍 언어와 제 *4*장, *GNOME* 코어 라이브러리 사용하기에서 사용한 것과 동일한 설정을 사용할 것이다. 본문에서 생성하는 프로젝트는 Vala GTK+(간단한) 프로젝트이므로 **GtkBuilder support for the user interface** 를 비활성화하고 **No license** 옵션을 활성화한다.

위젯의 표준 함수 오버라이드하기

보통은 위젯의 프로퍼티에서 기본값을 다른 값으로 대체함으로써 위젯이 어떻게 행동하는지, 위젯이 어떻게 표시되는지를 맞춤설정한다. 위젯을 한 두 번 맞춤설정해서 끝날 일이라면 전혀 문제가 되지 않는다. 그저 객체로 인스턴스화하여 프로퍼티를 맞춤설정하면 된다. 하지만 위젯의 재사용을 원한다면 위젯을 우선 서브클래싱하여 서브클래스를 맞춤설정해야 한다. 서브클래싱이란 기존의 특정 위젯 클래스를 기반으로 새로운 위젯 클래스를 생성함을 의미한다.

실행하기 - set_title 함수 오버라이드하기

이제 GtkWindow를 서브클래싱하고, GtkWindow의 set_title 함수의 행위를 변경하길 원한다고 가정하자. 함수는 창의 제목을 인자에서 전달되는 특정 문자열로 설정하는 데에 사용된다. 본문에서는 우리가 set_title 함수를 설정할 때마다 창의 제목으로 특별한 단어를 추가하는 새로운 행위를 소개하고자 한다.

1. custom-overriding이라는 새로운 Vala 프로젝트를 생성하라.
2. src/custom_overriding.vala 파일에 아래 코드를 이용하라.

```
using GLib;
using Gtk;

public class CustomWindow : Window
{
    public CustomWindow ()
    {
    }

    public new void set_title(string newTitle)
    {
        title = "Custom: " + newTitle;
    }
    static int main (string[] args)
    {
        Gtk.init (ref args);
        var window = new CustomWindow ();
        window.set_title ("My window");
        window.show_all ();
        window.destroy.connect(Gtk.main_quit);

        Gtk.main ();

        return 0;
    }
}
```

3. JavaScript에 상응하는 코드는 아래와 같다. 이것을 custom-overriding.js라는 스크립트 파일에 넣어라.

```
#!/usr/bin/env seed

Gtk = imports.gi.Gtk;
GObject = imports.gi.GObject;

CustomWindow = new GType({
    parent: Gtk.Window.type,
    name: "CustomWindow",
    class_init: function(klass, prototype) {
        prototype.set_title = function(newTitle) {
            this.title = "Custom: " + newTitle;
        }
    },
});

Gtk.init(Seed.argv);
var window = new CustomWindow();
window.set_title("My window");
window.signal.destroy.connect(Gtk.main_quit);
window.show_all();
Gtk.main();
```

4. 위를 실행하고 아래 스크린샷과 같이 이름 앞에 **Custom: word** 가 자동으로 붙는지 확인하라.

무슨 일이 일어났는가?

Vala 코드를 먼저 살펴보자.

아래 선언을 통해서는 CustomWindow라는 새로운 클래스가 있으며 이는 Window에서 파생되었음을 알린다.

```
public class CustomWindow : Window
```

아래 행을 이용해 Gtk와 Glib 네임스페이스를 사용 중이므로 GtkWindow를 명시적으로 나타낼 필요는 없다.

```
using GLib;
using Gtk;
```

앞의 코드를 이용하면 Window의 전체 이름이 Gtk.Window이거나 Glib.Window가 될 가능성이 있다. 하지만 Glib.Window라는 것은 존재하지 않기 때문에 Gtk.Window에서 새로운 CustomWindow 클래스를 서브클래싱하고 있음을 꽤 확신할 수 있다.

이제 빈 생성자가 있는데, 이는 비어 있기 때문에 원한다면 생략해도 좋다.

```
public CustomWindow ()
{
}
```

아래는 set_title 이라 불리는 함수를 선언한다. 여기서는 set_title 함수를 우리 함수로 오버라이드한다.

```
public void set_title(string newTitle)
{
    title = "Custom: " + newTitle;
}
```

오버라이드가 가능하려면 함수 인자(메서드 시그니처라고 불리기도 함)가 정확히 같아야 하는데, 같지 않을 경우 Vala는 아래 행을 이용해 우리가 함수를 호출하면 경고 메시지를 발생시킬 것이다.

```
window.set_title ("My window");
```

이 함수는 원본 Gtk.Window 클래스의 set_title 함수 대신에 호출된다. set_title 함수의 본체 내에서 Custom: 문자열로 title 프로퍼티의 값을 설정하고, 함수의 인자로 제공되는 newTitle과 결합(concatenate)한다.

애플리케이션에서 이를 다시 사용하고자 한다면 일반적인 Window 클래스 대신 아래 코드를 이용해 CustomWindow 클래스를 인스턴스화할 필요가 있다.

```
var window = new CustomWindow ();
```

그러면 새로 정의된 창 객체는 set_title 함수에 설정된 새로운 행위 뿐만 아니라 Gtk.Window로부터 비롯된 모든 기능들을 갖게 될 것이다.

Seed를 이용하면 아래 JavaScript 코드를 이용함으로써 set_title에 새로운 행위를 정의한다.

```
class_init: function(klass, prototype) {
    prototype.set_title = function(newTitle) {
        this.title = "Custom: " + newTitle;
    }
}
```

이는 제 4장, GNOME 코어 라이브러리 사용하기에서 init 함수에 직접 새로운 함수를 넣을 때 학습한 내용과는 약간 다르다. 이번 장에서는 개념적으론 차이가 있지만 제 3장, 프로그래밍 언어에서 보인 바와 약간 비슷하게 prototype 안에 함수를 넣는다.

사실상 init 함수 안에서 직접 사용하는 수도 있다. 하지만 오버라이드된 함수인지 명확히 식별하고 가독성을 향상시키기 위해선 선언을 class_init의 prototype 매개변수로 넣어야 한다. 이는 class_init 함수 내에 가상 함수(짧게 말해, 파생된 클래스에서 오버라이드할 수 있는 함수들)로 연결하는 C-언어 코드에서 GObject가 작동하는 방식과 유사하다.

이것이 바로 JavaScript에서 행하는 방식인데, 제 4장의 GNOME 코어 라이브러리 사용하기에서 논한 바와 같이 언어 자체는 실제 OOP 언어가 아니기 때문이다. 사실 우리는 동시에 두 개의 장소에서 함수를 정의할 수도 있지만 init 함수에 정의된 함수만 사용될 것이다. 이는 class_init에서 생성된 함수는 모두 클래스가 생성될 때 생성되고, init에서 생성된 함수는 객체가 인스턴스화될 때 생성되기 때문이다. 따라서 class_init에 선언하는 함수들은 후에 객체가 생성되면 init의 함수들과 동일한 이름을 갖기 때문에 init에서 오버라이드된다.

새로운 기능 추가하기

일부 오래된 기능의 새로운 행위 뿐 아니라 위젯에 새로운 기능을 추가하길 원한다고 가정해보자. 라이브 검색을 실행하기 위한 내부 텍스트 엔트리를 갖고 있는 창을 구현한다고 치자. 해당 위젯은 키보드의 어떤 키든 누르는 즉시 텍스트 엔트리를 표시할 것이다. 텍스트 엔트리 내의 값은 후에 애플리케이션의 다른 개체(entity)가 라이브 검색을 실행 시 사용할 것이다.

실행하기 - 컴포짓 위젯 만들기

앞서 언급했듯이 창은 하나의 자식만 가질 수 있는데, 그렇다면 창의 실제 내용과 텍스트 엔트리는 어떻게 추가할 것인가? 이를 논해보자.

1. custom-composite.js라고 불리는 스크립트를 생성하고 아래 코드로 채워라.

```
#!/usr/bin/env seed

Gtk = imports.gi.Gtk;
GObject = imports.gi.GObject;

CustomWindow = new GType({
    parent: Gtk.Window.type,
    name: "CustomWindow",
    signals: [
            {
                name: "search-updated",
                parameters: [GObject.TYPE_STRING]
            }
        ],

    class_init: function(klass, prototype) {
        prototype.show_search_box = function() {
            this.entry.show();
            this.entry.has_focus = true;
        }
        prototype.hide_search_box = function() {
            this.entry.hide();
        }
        prototype.super_add = prototype.add;
        prototype.add = function(widget) {
            if (widget != this.box) {
                this.box.pack_start(widget, true, true);
```

```
            } else {
                this.super_add(widget);
            }
        }
    },
    init: function(self) {
        this.box = new Gtk.Box();
        this.box.orientation = Gtk.Orientation.VERTICAL;
        this.entry = new Gtk.Entry();
        this.add(this.box);
        this.box.pack_start(this.entry, false, true);
        this.box.show();

        this.entry.signal.key_release_event.connect(function(obj,
event) {
            self.signal.search_updated.emit(self.entry.text);
            return false;
        });

        this.signal.key_press_event.connect(function(obj, event) {
            if (!self.entry.get_visible()) {
                self.show_search_box();
            }
            return false;
        });
    }
});

Gtk.init(Seed.argv);
var window = new CustomWindow();
var label = new Gtk.Label({label:'This is a text'});
window.add(label);
window.resize(400, 400);

window.signal.connect('search-updated', function(object, value) {
    label.set_text('Searching for keyword: ' + value);
});

label.show();
window.show();

Gtk.main();
```

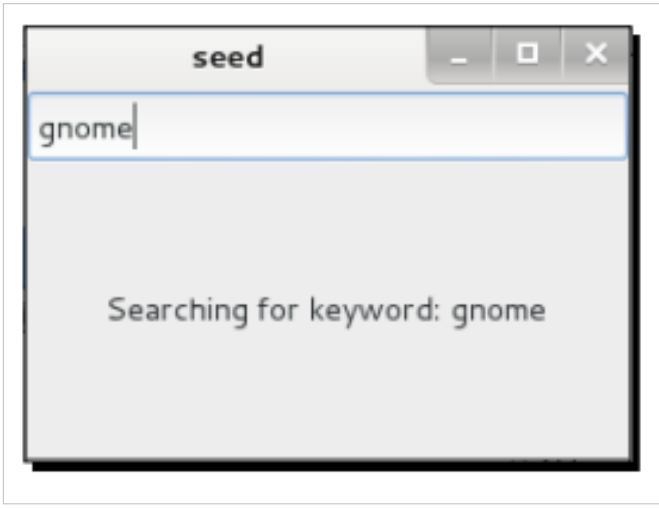
2. 아니면 custom-composite이라는 Vala 프로젝트를 생성하여 src/custom_composite.vala를 아래와 같은 코드 조각으로 채울 수도 있다.

```
using GLib;
using Gtk;
```

```
public class CustomWindow : Window
{
    Entry entry;
    Box box;
    public signal void search_updated(string value);

    void show_search_box() {
        entry.show();
        entry.has_focus = true;
    }

    void hide_search_box() {
        entry.hide();
    }

    public override void add(Widget widget) {
        if (widget != box) {
            box.pack_start(widget, true, true);
        } else {
            base.add(widget);
        }
    }

    public CustomWindow ()
    {
        box = new Box(Orientation.VERTICAL, 0);
        entry = new Entry();
        box.pack_start (entry, false, true);
        box.show();

        add(box);

        key_release_event.connect((event) => {
            search_updated(entry.text);
            return false;
        });

        key_press_event.connect((event) => {
            if (!entry.get_visible()) {
                show_search_box();
            }
            return false;
        });
    }

    static int main (string[] args)
```

```
    {
        Gtk.init (ref args);
        var window = new CustomWindow();
        var label = new Label("This is a text");

        window.add(label);
        window.resize(400,400);
        window.search_updated.connect((value) => {
            label.set_text("Searching for keyword " + value);
        });

        label.show();
        window.show();
        Gtk.main ();

        return 0;
    }
}
```

3. 프로그램을 실행하고 아래 스크린샷과 같이 입력하라.



무슨 일이 일어났는가**?**

방금 컴포짓 위젯, 즉 몇 개의 위젯으로 구성된 위젯을 생성하였다. GtkWindow, GtkBox, GtkEntry를 이용해 새로운 CustomWindow 위젯이 만들어졌고, 애플리케이션에 노출되는 인터페이스는 GtkWindow에서 비롯된 인터페이스가 유일하다. 이는 우리가 확장하고자 하는 위젯이 GtkEntry 클래스 또는 GtkBox 클래스가 아니라 GtkWindow 클래스이기 때문이며, 그에 따라 GtkWindow를 기반 클래스로 선택한다.

사용자가 텍스트 엔트리에 무언가를 입력하면 트리거되는 새로운 시그널도 추가한다. 이 시그널은 텍스트 엔트리의 내용을 출력하므로 검색 엔진과 같은 곳에 데이터를 제공할 때 유용하게 사용된다.

JavaScript 코드를 먼저 살펴보자.

```
CustomWindow = new GType({
    parent: Gtk.Window.type,
    name: "CustomWindow",
```

앞의 클래스 정의는 새로운 CustomWindow 클래스가 GtkWindow의 서브클래스이며 CustomWindow로 명명함을 Seed로 알려준다.

이제 search-updated라는 새로운 타입 문자열의 시그널을 선언한다.

```
     signals: [{
        name: "search-updated",
        parameters: [GObject.TYPE_STRING]
            }],
```

이 시그널을 이용해 텍스트 엔트리는 추후 처리에 사용 가능한 새로운 데이터를 포함하고 있음을 알려준다.

전체적인 이야기를 더 쉽게 알리기 위해 init 함수로 먼저 시작해보자.

```
    init: function(self) {
        this.box = new Gtk.Box();
        this.box.orientation = Gtk.Orientation.VERTICAL;
        this.entry = new Gtk.Entry();
```

여기서는 창의 두 개의 지원 객체, GtkBox와 GtkEntry를 init에서 선언한다. 이 시점에서 CustomWindow 객체가 생성될 예정이다. 이 객체를 class_init로 넣어선 안 되는데, class_init 함수는 클래스가 생성될 때 한 번만 호출되기 때문이다. 우리는 현재 CustomWindow 객체가 생성될 때 두 개의 지원 객체가 생성되길 원하기 때문에 init 함수에 넣는다.

다시 돌아와서 GtkEntry는 CustomWindow 에 넣길 바란다. 하지만 하나의 자식 위젯만 가질 수 있다는 창의 한계점 때문에 CustomWindow 클래스에서 컨테이너의 역할을 인수하도록 GtkBox를 사용한다. 위젯을 수직적으로 배치하고 GtkEntry를 가장 위에 놓는다.

아래 행을 이용해 박스를 창에 직접 추가한다.

```
        this.add(this.box);
```

그렇다면 애플리케이션에서 또 다른 위젯을 창으로 추가하길 원한다면 어떤 일이 발생할까? CustomWindow 내용은 이미 box 객체로 채워져 있지 않은가? 맞다. 그렇다면 CustomWindow 클래스에 위젯을 넣는 대신 box 객체로 추가하는 무언가를 생각해내야 한다. 이것은 add 함수를 재정의함으로써 class_init에서 실행하고자 한다. 간략하게 살펴보겠다.

다음은 GtkEntry 클래스를 패킹하고 box 객체를 표시할 차례다.

```
this.box.pack_start(this.entry, false, true);
this.box.show();
```

아래 코드 조각에서는 key-release-event 이벤트 핸들러를 추가한다. 따라서 키 누름을 해제할 때마다 텍스트 엔트리에서 일부 데이터를 이용할 수 있음을 표시하는 시그널을 발생시킨다.

```
this.entry.signal.key_release_event.connect(function(obj, event) {
        self.signal.search_updated.emit(self.entry.text);
        return false;
    });
```

우리는 텍스트 엔트리의 내용을 전달함으로써 시그널을 트리거한다. 성능과 관련된 이유로 키 누름 이벤트에서 실행하는 대신 키 누름해제 이벤트에서 실행한다. 무언가를 입력할 때는 타이핑이 끝나야, 즉 키 누름이 해제되어야만 텍스트가 준비되었음을 알 수 있다. 키 누름 이벤트에서 시그널을 트리거할 경우, 가령 다수의 중복된 문자를 텍스트 엔트리로 입력하기 위해 키를 길게 누른다 치면 시그널은 계속해서 발생할 것인데, 이 때 그다지 효과적이지 않은 검색 엔진을 사용 중이라면 이 과정에서 애플리케이션의 속도를 저하시킬 수도 있다.

여기서는 애플리케이션에서 키를 누를 때마다 초기에 텍스트 엔트리를 표시하도록 key-press-event 이벤트를 사용한다.

```
this.signal.key_press_event.connect(function(obj, event) {
    if (!self.entry.get_visible()) {
        self.show_search_box();
    }
    return false;
});
```

메서드 선언을 포함하는 class_init로 이동해보자.

```
class_init: function(klass, prototype) {
    prototype.show_search_box = function() {
        this.entry.show();
        this.entry.has_focus = true;
    }
    prototype.hide_search_box = function() {
        this.entry.hide();
    }
```

class_init 함수에 새로운 유틸리티 함수를 추가하므로, 후에 객체가 생성될 때는 함수가 준비되어 있을 것이다. 앞의 함수들은 텍스트 엔트리를 표시하고(키보드 포커스를 잡는 것을 포함해) 숨길 것이다. 아래는 GtkWindow의 add 함수를 수정하는 것과 관련된다.

```
prototype.super_add = prototype.add;
prototype.add = function(widget) {
    if (widget != this.box) {
        this.box.pack_start(widget, true, true);
    } else {
        this.super_add(widget);
    }
}
```

init 함수에서 보았듯이 박스를 CustomWindow 클래스에 추가한다. 앞의 함수에서는 특별한 처리 방법이 있기 때문에 추가된 위젯이 우리의 box 객체일 경우 그저 기반 클래스에서 함수를 호출하면 되는데, 기본적으로 이 함수는 GtkWindow의 원본 add 함수에 해당한다. 이를 위해서는 먼저 super_add 변수에 원본 함수를 저장해야 한다. 함수를 저장하고 나면 추가된 위젯을 박스로 패킹함으로써 add 함수를 재정의한다.

애플리케이션에서 새로운 CustomWindow 위젯을 활용하는 방법을 살펴보자. 아래 코드를 통해 CustomWindow 클래스로부터 새로운 객체를 선언한다.

```
Gtk.init(Seed.argv);
var window = new CustomWindow();
```

이후 새로운 라벨을 생성하여 창으로 추가한다. 여기서 라벨은 내부 박스 객체로 패킹됨을 주목한다.

```
var label = new Gtk.Label({label:'This is a text'});
window.add(label);
```

그리고 아래의 행이 있다.

```
window.resize(400, 400);
```

앞의 코드 행에서는 resize 함수를 선언하진 않았지만 기반 클래스, 즉 GtkWindow 클래스에서 그냥 이용할 수 있음을 알 수 있다. 따라서 꼭 GtkWindow에서 선언될 필요는 없으며, GtkWindow 클래스의 부모 클래스, 심지어 GtkWindow 클래스의 부모의 부모 클래스에서 선언되어도 괜찮다는 뜻이다.

아래 코드는 새로운 시그널을 사용하는 방법을 보여준다.

```
window.signal.connect('search-updated', function(object, value) {
    label.set_text('Searching for keyword: ' + value);
});
```

시그널이 발생할 때마다 우리는 라벨 안에 텍스트 엔트리의 내용을 표시한다. 실제 애플리케이션에서는 검색 엔진으로 값을 제공할 것인데, 데이터베이스 내 특정 데이터를 검색하거나 문서에서 단어를 검색하는 방법 중 택할 수 있다.

다음으로 Vala 코드를 분석해보자. 여기서는 먼저 CustomWindow 클래스가 Window 클래스로부터 서브클래싱된다고 말한다.

```
public class CustomWindow : Window
```

entry 와 box 위젯을 아직까지 건드리지 않았음을 기억하고, 이제 이들을 선언한다.

```
Entry entry;
Box box;
```

이후 새로운 시그널을 선언한다. Vala에서는 시그널에 대한 단어 구분자로 대시(dash) 대신 밑줄을 사용함을 기억하라.

```
public signal void search_updated(string value);
```

add 함수를 여기에서 정의하고, 위젯이 박스가 아닐 경우 위젯을 패킹한다.

```
public override void add(Widget widget) {
    if (widget != box) {
        box.pack_start(widget, true, true);
    } else {
        base.add(widget);
    }
}
```

위젯이 박스가 맞다면 원본(original) GtkWindow 클래스의 add 함수를 사용한다. 간단히 base.add() 를 사용하고, JavaScript 코드처럼 별도로 번거로운 작업을 하지 않아도 된다.

이 함수가 부모 클래스로부터 원본 add 함수를 오버라이드한다는 사실을 나타내기 위해 override 키워드를 사용한다. 해당 키워드가 없이는 함수는 완전히 새로운 함수이자 부모 클래스의 add 함수와 연관되지 않은 함수로 간주된다.

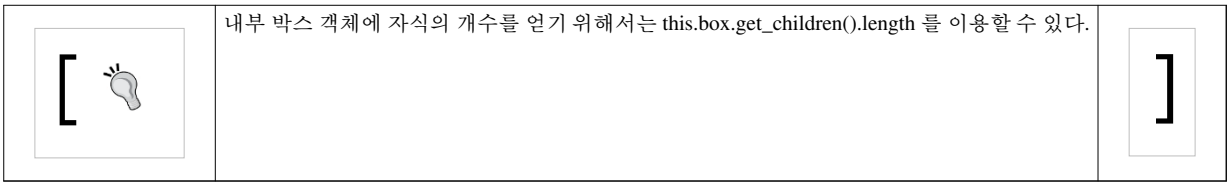
나머지 코드 부분은 꽤 간단하고 JavaScript 코드에서 앞서 실행한 것과 유사하다.

호환성 유지하기

CustomWindow 클래스를 자세히 살펴보면 GtkWindow와 호환성을 어긴다는 사실을 확인할 수 있다. GtkWindow는 하나의 자식 위젯만 취할 수 있지만 우리가 사용한 CustomWidget 클래스는 하나 이상의 위젯을 취할 수 있는데, 위젯들이 우리의 내부 박스로 패킹될 것이기 때문이다. GtkWindow와 호환성을 유지하기 위해서는 CustomWindow 클래스 내부에 이미 하나의 자식이 있을 때마다 잇따른 추가 요청을 거부함으로써 이러한 문제를 해결할 수 있다.

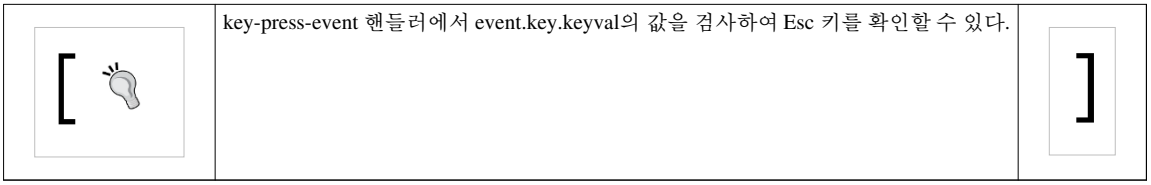
remove 함수를 재정의해야 한다는 것도 잊어버렸기 때문에 자식 위젯을 제거하고자 할 때마다 실패할 것인데, 그 이유는 자식 위젯이 CustomWindow 클래스가 아니라 내부 box 객체로 저장되기 때문이다. GtkWindow 클래스의 remove 함수는 이를 거부할 것인데, 이는 자식의 부모가 (내부 박스) 더 이상 같지 않기 때문이다 (CustomWindow).

이 문제는 해결이 가능하다. 첫 번째 문제의 경우, 내부 box 객체에 있는 자식의 개수를 확인할 수 있다. 새로운 위젯을 추가했을 때 자식의 개수가 2개(GtkEntry 클래스와 자식)를 넘으면 요청을 거부한다. 두 번째 문제에 대해서는 remove 함수를 재정의하고 내부 box 객체로부터 자식 위젯을 제거하면 된다.



시도해보기 - 엔트리 사용 후 숨기기

hide_search_box 함수를 준비해봤지만 아직 사용한 적은 없다. 한 가지 좋은 생각이 있다. Esc 키를 누르면 검색 박스를 숨겨보는 것이 어떤가?



GTK+ 커스텀 위젯 구현하기

다음 단계는 현재 존재하는 위젯으로부터 확장하는 것이 아니라 처음부터 새로운 위젯을 생성함으로써 커스텀 위젯을 구현해볼 차례다. GTK+ 표준 위젯에서 비슷한 위젯을 찾을 수 없을 때 이 방법을 이용하면 우리가 구현하고자 하는 위젯에서 필요로 하는 바를 성취할 수 있다.

이는 Vala에서만 가능한 일인데, 안타깝게도 본 서적을 집필하는 동안 사용된 Seed의 버전은 클래스로부터 호출될 때 함수의 오버라이드를 적절하게 처리할 수 없다.

본문에 소개된 예제는 사실상 방금 논한 실제 상황에는 영향을 미치지 않는데, 처음부터 위젯을 구현하는 대신 쉽게 사용이 가능한 위젯을 찾는 일이 더 간단하기 때문이다. 하지만 예제를 제공한 이유는 새로운 위젯을 구현하는 데에 수반되는 노력과 그 방법을 소개하기 위함이다.

아래의 요구사항을 충족하는 위젯이 필요하다고 가정해보자.

- 위젯은 직사각형 등의 데코레이션을 영역 내에 그릴 수 있어야 한다.
- 마우스로 위젯을 클릭하면 색상이 변하면서 누름 상태가 되었음을 나타낸다.
- 마우스 누름을 해제하면 색상이 일반 상태의 색으로 돌아간다.
- 마우스로 위젯을 클릭하면 위젯이 활성화되었음을 알려주는 시그널이 발생하고, 마우스 누름을 해제하면 이제 위젯이 비활성화되었음을 알려주는 시그널이 트리거된다.

실행하기 - 커스텀 위젯 구현하기

그렇다면 이제 앞에 소개한 디자인에 따라 구현해보자.

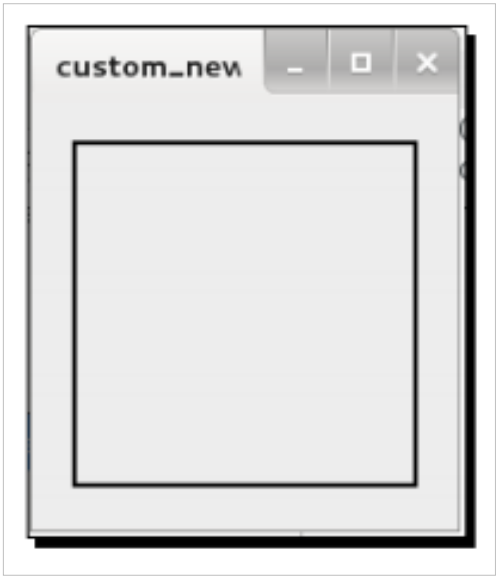
1. custom-new라는 새로운 Vala 프로젝트를 생성하고, src/custom_new.vala를 편집하여 아래 코드로 채워라.

```
using GLib;
using Gtk;

public class CustomWidget : DrawingArea
{
```

```
    StateFlags state;
    const int MARGIN = 20;
    public signal void activated();
    public signal void deactivated();

    void update_state (int newState)
    {
        switch (newState) {
        case 1: state = StateFlags.SELECTED;
            break;
        case 0:
        default:
                state = StateFlags.NORMAL;
            break;
        }
        queue_draw ();
    }

    public override bool draw(Cairo.Context cr) {
        StyleContext style = get_style_context ();
            style.set_state (state);

        int w = get_allocated_width ();
        int h = get_allocated_height ();

            Gtk.render_background (style, cr, 0, 0, w, h);

        cr.rectangle (MARGIN, MARGIN,
                    w - (MARGIN * 2),
                    h - (MARGIN * 2));
        cr.stroke ();
        return true;
    }

    public CustomWidget()
    {
        update_state (0);
        add_events (Gdk.EventMask.BUTTON_PRESS_MASK
                | Gdk.EventMask.BUTTON_RELEASE_MASK);

        button_press_event.connect((e) => {
            update_state (1);
            activated();
            return true;
        });

        button_release_event.connect((e) => {
```

```
            update_state (0);
            deactivated();
            return true;
        });
    }

    static int main (string[] args)
    {
        Gtk.init (ref args);
        var window = new Window();
        var widget = new CustomWidget();

        window.add (widget);
        window.show_all ();
        Gtk.main ();

        return 0;
    }
}
```

2. 프로그램을 실행하고 앞서 명시한 요구사항을 테스트하라.



무슨 일이 일어났는가**?**

코드를 보면 처음부터 자신만의 위젯을 구현하기란 꽤 수월해 보인다. 자세히 살펴보자.

```
public class CustomWidget : DrawingArea
```

클래스 선언을 보면 GtkDrawingArea를 기반 클래스로 사용 중임을 확인할 수 있다. GtkDrawingArea 클래스는 빈 위젯으로, 아무 일도 하지 않고 아무 것도 표시하지 않는다. 이 위젯의 특징은 그저 그 위에 무언가를 그릴 수 있다는 것이다. GTK+ 용어로 말하자면 이 위젯은 그리기가 가능하다(drawable)고 말할 수 있겠다. GtkDrawingArea를 선택한 이유는 우리의 요구사항에 일치하기 때문이며, 개발자들이 처음부터 새로운 위젯을 구현할 때 사용하는 일반적인 위젯이기 때문이다.

```
StateFlags state;
```

아래는 위젯의 누름 상태를 보유하게 될 변수다.

```
const int MARGIN = 20;
```

이는 2.0의 값을 보유하는 MARGIN이라는 상수다. 위젯의 변으로부터 직사각형의 여백으로 사용될 것이다. 전체적인 GNOME 프로젝트에서는 상수의 이름을 모두 대문자로 작성하는 규칙이 사용되므로 GNOME 애플리케이션 개발을 할 때는 항상 이러한 규칙을 따르는 편이 좋다.

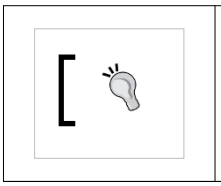
위젯에서 제공하는 두 개의 시그널은 다음과 같다.

```
public signal void activated();
    public signal void deactivated();
```

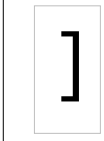
이는 위젯의 상태를 저장한 후 queue_draw를 호출하는 유틸리티 함수다.

```
void update_state (int newState)
{
    switch (newState) {
    case 1: state = StateFlags.SELECTED;
        break;
    case 0:
    default:
        state = StateFlags.NORMAL;
        break;
    }
    queue_draw ();
}
```

state 변수는 기본적으로 상태(state)에 대한 고유의 해석을 보유하여 위젯의 조건(condition)으로 매핑한다. 요구조건을 다시 살펴보면, 위젯을 클릭하거나 위젯으로부터 마우스를 해제하면 색상이 변경되길 원한다. 여기서 색상을 상태와 매핑한다. 어떤 것도 누르지 않으면(혹은 마우스 누름을 해제하면) 상태를 NORMAL 상태로 설정한다. 반대로 마우스를 누르면 상태를 SELECTED 상태로 선택한다. 이 두 가지는 NORMAL, SELECTED, ACTIVE, INSENSITIVE 등 여러 상태가 포함된 열거의 일부에 해당한다.



여러 상태에 대한 정보는 DevHelp의 GtkStateFlags에서 확인할 수 있다.



이러한 상태를 마음대로 매핑해도 좋지만 매핑할 때는 상식을 이용해야 한다. 즉, 위젯이 눌러지면 우리는 위젯이 사용자와 상호작용을 할 수 없거나 비활성화되었다고 생각하는 대신 위젯이 활성화 또는 선택되었거나 포커스를 받았다고 생각할 것이다. 그리고 이전 플래그를 이용해 이런 가상의 상태에 가장 가까운 의미를 찾을 것이다. 따라서 누름 상태의 위젯에서는 SELECTED 를 이용해 매핑하고, 눌러지지 않은 위젯은 NORMAL 상태로 매핑한다.

함수의 끝에 보면 위젯이 그림을 새로고침하도록 만들기 위해 queue_draw를 호출한다. 기본적으로는 GTK+에게 가능한 한 빨리 draw 함수를 호출할 것을 요청하는 셈이다.

위젯의 시각적 모양을 책임지는 draw 함수의 모습은 다음과 같다.

```
public override bool draw(Cairo.Context cr) {
```

이 함수에서는 위젯에 어떤 것이든 그릴 수 있다. 인수에서 전달되는 시스템에서 얻는 것은 Cairo Context 객체다. Cairo는 GTK+가 위젯을 그리기 위해 사용하는 벡터 기반의 캔버스 시스템이다. Context는 GKT+에 의해 생성된 핸들 객체로, 우리는 이를 이용해 무엇을 그릴지, 어디에 그릴지 등을 제어할 수 있다. 이는 HTML5의 캔버스와 비슷하다.

요구조건을 다시 살펴보면, 상태에 따라 위젯의 색상을 채워야 한다고 언급되어 있다. 그렇다면 어떤 색상을 이용해야 할까? 우리는 위젯에 직접적으로 색상을 선언할 필요가 없다(그래선 안 된다!). 대신 GTK+ 테마 시스템(theming system)에 의존해야 한다.

이 함수에서 가장 먼저 하는 일은 스타일 컨텍스트를 얻는 일이다.

```
StyleContext style = get_style_context ();
    style.set_state (state);
```

컨텍스는 무엇보다 특정 상태에서 사용되는 패턴이나 색상을 제어한다. 이는 update 함수에서 설정하는 상태다. 이후 상태 정보를 이용해 컨텍스트를 설정한다. 그리고 나면 스타일 컨텍스트는 명시된 상태로 자동 전환할 것이다.

render_background 함수를 호출하면 위젯 내 명시된 영역을 색상 또는 패턴으로 채울 것이다.

```
int w = get_allocated_width ();
int h = get_allocated_height ();
    Gtk.render_background (style, cr, 0, 0, w, h);
```

이런 경우 0,0 좌표부터 시작해 위젯에 할당된 전체 너비와 높이 크기만큼 색상 또는 패턴으로 모두 채운다. 따라서 색상 또는 패턴이 위젯 전체를 채울 것이라고 기대한다. 실제 사용되는 색상이나 패턴은 GTK+ 테마에 명시된다. 테마가 NORMAL 상태의 색상이 파란색이라고 말하면 GTK+는 파란색으로 위젯을 채울 것이다.

요구조건에 따라 배경을 그리고 나면 직사각형으로 전경(foreground)을 그려야 한다. 이 때는 단순히 MARGIN의 값, 즉 20 픽셀 가까이 축소된 크기의 위젯으로 직사각형을 그리면 된다. 직사각형은 위젯 영역 내에 그려질 것으로 예상한다.

```
cr.rectangle (MARGIN, MARGIN,
    w - (MARGIN * 2),
    h - (MARGIN * 2));
```

Cairo에서는 많은 명령을 제공할 수 있지만 실제로 이러한 명령을 실행하기 전에는 어떤 것도 렌더링되지 않는다. 앞에서 소개한 rectangle 명령과 같은 테두리 그리기(drawing stroke) 명령의 경우 캔버스로 그리기를 실행하려면 stroke 함수를 이용한다.

```
cr.stroke ();
```

그러면 위젯에 직사각형이 그려진다.

함수로부터 true 값을 리턴하는 것은 GTK+ 시스템의 다른 부분에서 더 이상 어떠한 처리도 필요로 하지 않는다는 뜻이다.

```
return true;
```

그렇다면 draw 함수를 호출하는 것은 무엇일까? 많은 요인들이 있는데, 앞서 논한 queue_draw도 그 중 하나다. 위젯을 포함하는 창의 크기를 조정한다거나, 또 다른 창이 위젯의 일부를 가리는 경우도 해당할 수 있겠다. 구체적으로 말하자면, GTK+가 위젯을 다시 그려야 한다고 판단하게 만드는 것이라면 무엇이든 draw 함수를 트리거할 것이다. draw 함수는 항상 반복해서 호출 가능하기 때문에 함수의 내용은 가능한 한 작게 주의하여 디자인해야 한다. 위젯의 만족스러운 시각적 모양을 얻기 위해서는 draw 함수에서 느리게 처리되는 복잡한 계산이나 호출 함수는 피해야 한다. 가령 위젯이 애니메이션을 표시할 경우 부드럽고 유체적인 애니메이션을

얻기 위해서는 60 frames per second(fps)의 프레임률을 가져야 한다. 즉, 초당 draw 함수가 60회 호출될 것으로 예상된다는 의미다. 이는 draw 함수가 16.67 밀리초 이내에 실행되어야 한다는 뜻이기도 하다. 이를 충족할 수 없다면 애니메이션은 버벅거리고 사용자에게 불쾌한 경험을 유발하기도 한다.

이제 생성자를 살펴보도록 하자.

```
public CustomWidget()
{
    update_state (0);
```

여기서 state 변수를 0으로 초기화하는데, 마우스 누름이 처음에 발생하지 않았기 때문에 위젯이 NORMAL 상태로 그려질 것으로 예상된다는 뜻이다.

아래 코드는 생성자 내에서 호출되어야 한다. 이는 버튼 누름 및 누름해제 이벤트가 발생할 때마다 위젯으로 이벤트를 전달할 것을 GTK+ 시스템으로 요청한다.

```
add_events (Gdk.EventMask.BUTTON_PRESS_MASK
    | Gdk.EventMask.BUTTON_RELEASE_MASK);
```

이를 실행하지 않으면 위젯은 누름 상태인지 아닌지를 알지 못하기 때문에 아래 두 개의 이벤트 핸들러가 애초에 호출되지 않을 수도 있다.

```
button_press_event.connect((e) => {
    update_state (1);
    activated();
    return true;
});

button_release_event.connect((e) => {
    update_state (0);
    deactivated();
    return true;
});
```

이러한 이벤트 핸들러에서는 true를 리턴함으로써 처리가 최종적이며 GTK+가 이벤트 파이프라인에 있는 다른 위젯들에게 이벤트를 전달하는 것은 원치 않음을 나타낸다.

처음부터 위젯을 생성하는 방법(이번 예제에서 소개)과 기존 위젯으로부터 서브클래싱하는 방법(앞의 예제에서 소개)의 차이는 생성자에 있는 이벤트 구독과 draw 함수에 있음을 주목해야 한다. 사실 기존 위젯을 서브클래싱하면서 자신만의 그리기(drawing) 함수를 구현하는 수도 있다. 어떤 경우에서는 draw 함수에서 base.draw()를 호출함으로써 원본 그리기 함수와 자신만의 그리기 함수를 섞어 사용하기도 한다.

라이브러리에서 위젯 관리하기

많은 프로젝트에서 사용되는 커스텀 위젯을 몇 가지 갖고 있어서 그로부터 라이브러리를 만든다면 소스 코드를 복사하는 수고와 위젯의 수정 여부를 굳이 추적할 필요가 없을 것이다. 위젯을 라이브러리로 모으면 여러 프로젝트에 사용 가능한 하나의 라이브러리와 하나의 소스 코드를 유지할 수 있다.

제 3장, 프로그래밍 언어에서는 JavaScript 코드의 모듈화를 논한 바 있는데 이는 그 앞에 소개한 원칙과 개념적으로 동일하다. Vala를 이용하면 더 많은 수고를 필요로 하지만, 최종적으로 보면 라이브러리에 위젯을 모을 수 있다는 장점을 확보할 수 있다.

실행하기 - 라이브러리 생성하기

Vala에서 이것을 어떻게 실현하는지에 집중하자. 두 개의 프로젝트를 생성하는데, 하나는 라이브러리 예제용이고 나머지 하나는 그 라이브러리의 사용자를 위한 것으로, 아래 단계를 따라한다.

1. 먼저 custom-library라는 Vala 프로젝트를 생성하자.
2. src/custom_library.vala에서 custom-new 프로젝트의 custom_new.vala 파일로부터 코드를 채운다. 코드를 복사하고 붙여 넣은 후 아래와 같이 몇 가지만 조정한다.
   1. 클래스 선언을 네임스페이스 선언으로 자동 줄바꿈(wrap)하여 코드가 아래와 같은 모양이 되도록 한다.

```
namespace CustomWidget {
public class CustomWidget : DrawingArea
```

   2. 코드 끝에 닫는 중괄호를 넣는 것을 잊지 말라.
   3. 클래스로부터 static main 함수를 제거하라.
   4. Files dock를 이용해 src/Makefile.am 파일의 전체 코드를 아래의 코드로 대체하라.

```
AM_CPPFLAGS = \
    -DPACKAGE_LOCALE_DIR=\""$(localedir)"\" \
    -DPACKAGE_SRC_DIR=\""$(srcdir)"\" \
    -DPACKAGE_DATA_DIR=\""$(pkgdatadir)"\" \
    $(CUSTOM_LIBRARY_CFLAGS)

AM_CFLAGS =\
    -Wall\
    -g

lib_LTLIBRARIES = libcustomwidget.la

libcustomwidget_la_SOURCES = \
    custom_library.vala config.vapi

libcustomwidget_la_VALAFLAGS = \
    --pkg gtk+-3.0 --library=libcustomwidget -X -fPIC -X
    -shared -H custom_widget.h

libcustomwidget_la_LDFLAGS = \
    -Wl,--export-dynamic
```

3. ***Shift+F7*** 키 조합을 눌러 프로젝트를 만들어라. 오류가 없도록 하고, src 디렉터리에서 custom_widget.h 와 libcustomwidget.vapi 를 찾고 .libs 디렉터리에서 라이브러리 파일 집합을 찾아라. 라이브러리를 사용할 준비가 되었음을 뜻한다.
4. 앞서 생성한 라이브러리를 사용하는 방법에 대한 예로 custom-library-client라는 프로젝트를 하나 더 생성하라. src/custom_library_client.vala 를 아래 코드로 채워라.

```
using GLib;
using Gtk;
using CustomWidget;

public class Main : Object
{
    public Main ()
    {
        Window window = new Window();
```

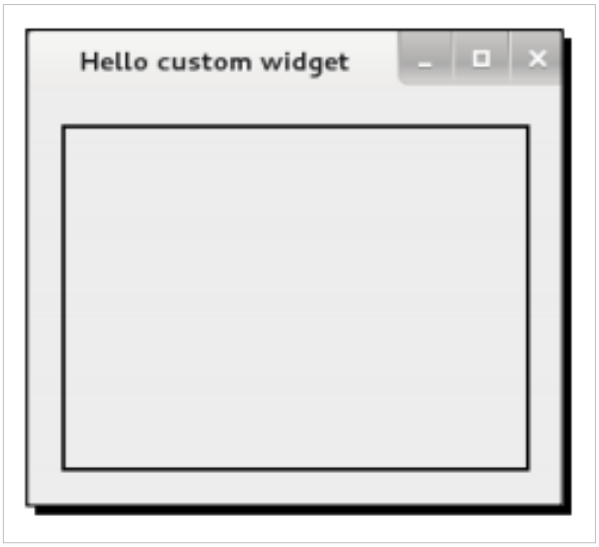
```
        var w = new CustomWidget.CustomWidget();
        window.set_title ("Hello custom widget");

        window.add(w);
        window.show_all();
        window.destroy.connect(on_destroy);
    }

    public void on_destroy (Widget window)
    {
        Gtk.main_quit();
    }

    static int main (string[] args)
    {
        Gtk.init (ref args);
        var app = new Main ();
        Gtk.main ();
        return 0;
    }
}
```

5. 프로젝트 내에 lib/ 디렉터리를 생성하고, custom-library 프로젝트로부터 src/custom_widget.h와 src/libcustomwidget.vapi 파일, 그리고 이름 앞에 .libs/libcustomwidget.so 가 붙은 파일을 모두 찾아라. 그 파일들을 방금 생성한 lib/ 디렉터리로 복사하라.
6. src/Makefile.am 을 아래의 내용으로 대체하라.

```
AM_CPPFLAGS = \
    -DPACKAGE_LOCALE_DIR=\""$(localedir)"\" \
    -DPACKAGE_SRC_DIR=\""$(srcdir)"\" \
    -DPACKAGE_DATA_DIR=\""$(pkgdatadir)"\" \
    $(CUSTOM_LIBRARY_CLIENT_CFLAGS)

AM_CFLAGS =\
    -Wall\
    -g -I../lib/

bin_PROGRAMS = custom_library_client

custom_library_client_SOURCES = \
    custom_library_client.vala config.vapi

custom_library_client_VALAFLAGS = \
    --pkg gtk+-3.0 --pkg libcustomwidget --vapidir ../lib

custom_library_client_LDFLAGS = \
    -Wl,--export-dynamic -lcustomwidget -L../lib
```

```
custom_library_client_LDADD = $(CUSTOM_LIBRARY_CLIENT_LIBS)
```

7. **Run** 메뉴를 열고 **Program Parameters** 를 선택하라. 대화상자가 나타나면 **Environment Variables** 박스를 확장시켜라.
8. LD_LIBRARY_PATH라는 새로운 엔트리를 생성하고 값 필드에 방금 생성한 lib/ 디렉터리의 전체 경로를 입력하라. 가령, 필자의 컴퓨터에서는 /home/gnome/src/ custom-libraryclient 에 custom-library-client 프로젝트가 있기 때문에 값 필드에 /home/gnome/src/ custom-library-client/lib라고 입력하였다.
9. 이제 애플리케이션을 실행할 수 있다. 앞의 예제와 비슷하지만 이번에는 창의 제목이 아래 스크린샷과 같이 표시된다.



무슨 일이 일어났는가**?**

라이브러리 프로젝트에서 클래스를 CustomWidget 네임스페이스로 감쌌다. 이 때문에 우리는 클라이언트 애플리케이션에 아래의 행을 넣을 수 있다.

```
Using CustomWidget;
```

라이브러리에서는 static main 함수를 제거한다. 라이브러리는 절대 main 함수를 가져선 안 되기 때문에 무조건 제거해야 한다. 애플리케이션의 진입점으로 정확히 하나의 main 함수만 존재해야 한다는 사실은 앞서 살펴본 바가 있는데, 이번 경우는 custom-library-client 프로젝트가 진입점이다.

이후 애플리케이션을 만드는 대신 라이브러리를 생성하고자 함을 Vala와 C 컴파일러로 알리기 위해 Makefile.am 을 수정하였다.

아래의 행은 libcustomwidget.la 의 이름으로 라이브러리를 생성하고자 하는 의도를 알려준다.

```
lib_LTLIBRARIES = libcustomwidget.la
```

실제로는 이로 인해 libcustomwidget.la와 libcustomwidget.so 라이브러리 파일의 집합이 생성될 것이다.

이후 아래의 행을 통해 잇따른 파일들로부터 libcustomwidget 라이브러리가 빌드됨을 알린다.

```
libcustomwidget_la_SOURCES = \
    custom_library.vala config.vapi
```

아래 실린 코드는 Vala 코드를 C-언어 소스 코드로, 그리고 마침내 Binary 실행 파일로 컴파일 시 언급된 이름으로 라이브러리를 만들어 -fPIC -shared 를 C 컴파일러로 전달하고 싶음을 나타낸다 (-x 는 -x 다음에 언급된 플래그라면 무엇이든 C 컴파일러로 전달하기 위한 옵션이다).

```
libcustomwidget_la_VALAFLAGS = \
    --pkg gtk+-3.0 --library=libcustomwidget -X -fPIC -X -shared -H
custom_widget.h
```

앞의 두 플래그는 라이브러리를 생성할 때 가장 중요한 플래그로, 라이브러리를 공유할 수 있도록 해주고 메모리 내 어떤 위치에서든 로딩 가능하게 만들어준다 (**PIC** 는 위치 독립 코드**(Position-independent Code)**를 뜻한다). 마지막으로 custom_widget.h 라는 C 헤더 파일을 생성하였음을 컴파일러로 알린다. 이 파일은 클라이언트 프로젝트에서 C 컴파일러가 필요로 하는 파일이다.

다음으로, 이 프로젝트에는 main 함수가 없기 때문에 프로젝트를 실행하는 대신 빌드한다. 결과 파일들은 시스템 전체적으로 /usr/include(custom_widget.h의 경우), /usr/lib(libcustomwidget.so. * 파일의 경우), /usr/share/vala/vapi(libcustomwidget.vapi 파일의 경우) 중 한 곳으로 복사하거나, 클라이언트 프로젝트의 lib 디렉터리에 모두 복사해야 한다.

클라이언트 프로젝트로 넘어가자.

여기서 유일하게 흥미로운 부분은 바로 src/Makefile.am 파일에 있다.

--pkg libcustomwidget은 추가 라이브러리로 명시한다. 또 libcustomwidget.vapi는 ../lib 디렉터리에서 이용 가능할 것이라고 명시한다.

```
custom_library_client_VALAFLAGS = \
    --pkg gtk+-3.0 --pkg libcustomwidget --vapidir ../lib
```

Vala는 위의 코드가 없다면 .vapi 파일을 어디서 찾아야 할지 알 수 없지만 단, 개발자가 시스템 전체에 파일을 설치하는 경우는 제외된다. 아래의 파일 일부는 링커(linker)와 관련된 것으로, ../lib/libcustomwidget.so* 에서 찾을 수 있는 libcustomwidget 라이브러리를 이용해 심볼(symbol)을 결정한다.

```
custom_library_client_LDFLAGS = \
    -Wl,--export-dynamic -lcustomwidget -L../lib
```

애플리케이션을 실행하는 동안 LD_LIBRARY_PATH를 libcustomwidget.so.* 파일을 포함하는 전체 경로로 설정한다. 시스템 전체에 라이브러리가 설치되어 있다면 이럴 필요가 없다.

마지막으로 .vapi 파일, .h 파일, 라이브러리 파일을 전송하고, 애플리케이션이 쉽게 검색 및 사용이 가능하도록 시스템 전체적으로 설치해야 하는데, 이제 GTK+ 라이브러리를 매우 쉽게 사용할 수 있기 때문이다. 이러한 라이브러리 접근법을 이용 시 장점은, 문제가 되는 프로그래밍 언어로 라이브러리와 헤더 파일이 결합되어 있는 경우 Vala 프로그램만 라이브러리를 이용할 수 있는 것이 아니라 C 프로그램이나 다른 프로그램 언어들과 함께 사용할 수도 있다는 데에 있다.

요약

이번 장에서는 기존의 위젯을 확장하는 방법, 기존의 위젯을 새로운 위젯으로 결합하는 방법, 새로운 위젯을 생성하는 방법에 대해 학습하였다. 기존의 위젯에 기능을 추가하거나 그로부터 제거하기란 꽤 수월하다는 사실을 확인할 수 있었다. 또 위젯을 확장할 때는 위젯을 원본 인터페이스의 이전 버전과 호환이 가능하도록 만드는 것이 매우 중요하다는 사실도 알게 됐다.

커스텀 위젯을 생성하면서 Cairo 캔버스를 이용한 그림 그리기를 잠시 논한 바 있다. GTK+ 스타일링 API를 간략하게 다루면서 색상 입히기나 다른 페인트 스타일은 테마 시스템에 의존하고, 프로그램에서 하드코딩을 피할 수 있었다. 또 페인팅 함수를 최적으로 유지하면 성능을 양호하게 유지할 수 있음을 논했다.

마지막으로, 위젯의 생성은 라이브러리의 생성과도 결부된다. Vala에서 위젯을 생성하는 방법을 다루어보았다. 지금까지 경험을 토대로 이미 만들어진 위젯을 고를 뿐 아니라 새로운 위젯을 확장 및 생성하여 애플리케이션으로 결합함으로써 GNOME 애플리케이션을 개발하는 데에 더 자신감을 얻게 되었다.

다음 장에서는 GStreamer를 이용한 멀티미디어 프로그래밍을 논하고자 한다. 미디어 파일을 갖고 놀고 조작하기 위해 GStreamer 프레임워크를 활용하는 방법도 논할 것이다.

Notes

GNOME3ApplicationDevelopmentBeginnersGuide:Chap 07

제 7 장 멀티미디어 즐기기

멀티미디어 즐기기

멀티미디어 기능은 GNOME의 가장 강력한 장점에 속한다. 이러한 기능은 개발자들이 쉽게 멀티미디어 내용을 표시하도록 다양한 API를 제공한다. 구현할 수 있는 광범위한 범위의 애플리케이션 아이디어를 제시하고, 단순한 오디오/비디오 변환 툴, 음악이나 비디오 스트림 재생기, CCTV 감시, 모든 기능을 갖춘 교육 애플리케이션을 예로 들 수 있겠다.

이번 장에서는 GStreamer API를 이용해 오디오와 비디오의 내용을 표시함으로써 GStreamer의 기본적인 사용을 살펴보도록 하겠다. 그 내용을 구체적으로 소개하자면 다음과 같다.

- GStreamer 개념
- 오디오와 비디오 재생하기
- 필터를 스트림에 적용하기

이제 이 주제들을 자세히 살펴보도록 하자.

필요한 패키지

이번 장에서는 MPEG 코드라는 소프트웨어를 사용하는데, 기본 Linux 배포판 보관소에서 무료로 이용할 수 있는 것이 아니다. Fedora 사용자는 제3자의 보관소를 추가해야만 소프트웨어를 이용할 수 있을 것이다. Terminal에 아래의 명령을 입력하여 제3자 라이브러리를 추가한다.

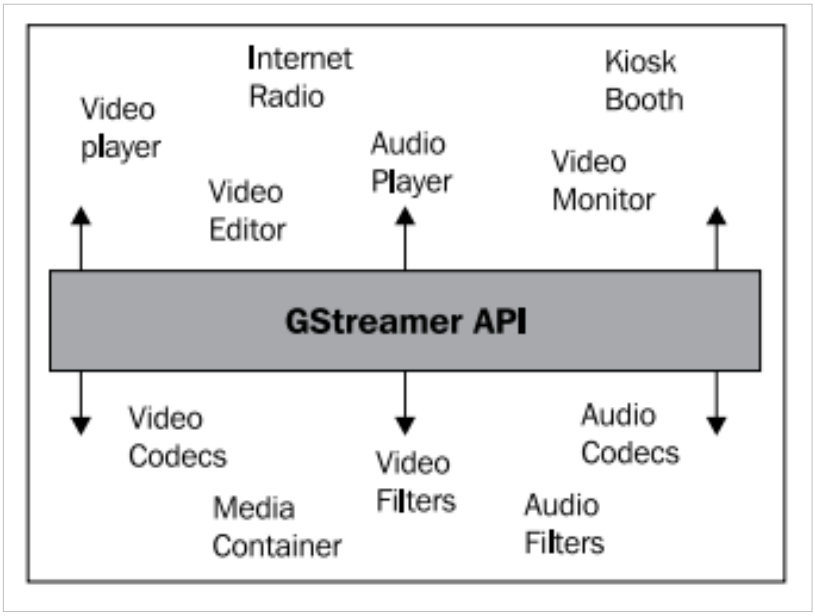
```
su -c 'yum localinstall --nogpgcheck
http://download1.rpmfusion.org/free/fedora/rpmfusion-free-release-stable.noarch.rpm

http://download1.rpmfusion.org/nonfree/fedora/rpmfusion-nonfree-release-stable.noarch.rpm'
```

아래 패키지가 설치되어 있도록 한다.

- **Fedora**: gstreamer-plugins-bad, gstreamer-plugins-ugly, gstreamer-ffmpeg, gstreamer-tools
- **Ubuntu/Debian**: gstreamer0.10-plugins-bad, gstreamer0.10-pluginsugly, gstreamer0.10-plugins-ffmpeg, gstreamer0.10-tools, libgstreamer-plugins-base0.10-dev

GStreamer의 기본 개념 이해하기

GStreamer란 GNOME이 멀티미디어 기능을 지원하기 위해 사용하는 미디어 처리 프레임워크다. 여기에는 미디어 스트림의 열기, 인코딩, 디코딩, 필터링에 사용되는 추상화 레이어(abstraction layer)를 제공하는 플러그인 구조(plugin infrastructure)가 있다. 즉, 특정 멀티미디어 포맷에 대한 플러그인이 있는 한 해당 포맷으로 파일을 열거나 쓰고 미디어를 재생하는 것이 가능하다는 의미다. 아래 그림은 GStreamer API의 아키텍처를 간소화한 모습이다.



그림에서 볼 수 있듯이 애플리케이션은 GStreamer가 제공한 APIs를 사용할 수 있고, 특정 코덱이나 필터의 구현에서도 API를 사용할 수 있다. 애플리케이션은 코덱이나 필터에 대한 세부적인 내용을 알 필요가 없으며, 코덱이나 필터 또한 애플리케이션의 세부 내용을 알 필요 없다. 불행하게도 실제 세계에서는 그림에 표시된 모든 것을 스스로 구현하는 동시 서로 차이가 있는 오디오 및 비디오 재생 애플리케이션들이 존재한다.

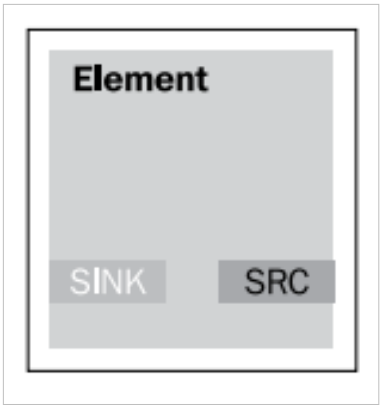
GStreamer에는 **"elements"**(요소)라는 개념이 있다. 이는 미디어 스트림에 수반되는 개체들의 기본적인 부분이다. 스트림이 시작되는 요소, 스트림이 끝나는 요소, 그리고 스트림이 전달되는 동안 스트림이 조작되는 추가적인 요소들도 있다.

데이터는 물로 생각할 수 있다. 요소들이 서로 파이프를 통해 연결된 시스템으로 물이 흐른다. 물은 요소에 의해 시스템으로 들어오는데, 물 갤런(water gallon), 물컵, 호흡기관, 또는 가상으로 산소와 수소 분자로부터 물을 생성하는 반응기까지 모두 이러한 시스템의 예가 된다.

색상이나 냄새가 필터 요소에 의해 변경될 수 있도록 시스템 내에서 물을 조작한다. 심지어 이러한 필터들을 결합할 수도 있다.

그리고 나면 물은 양동이, 물컵, 스프레이, 또는 증발기와 같은 요소를 통해 시스템에서 빠져나간다.

각 요소에는 최소한 하나의 문이 있는데, 문은 소스(source)나 싱크(sink), 혹은 둘 다 해당하기도 한다. **"Source"**(소스)는 데이터가 흐르기 시작하는 원점을 의미하고, **"sink"**(싱크)는 데이터가 흘러 들어가는 종점을 의미한다. 이러한 문을 **"pad"**(패드)라고 부른다. 각 요소는 하나 이상의 패드를 가질 수 있는데, 오디오 스트림과 비디오 스트림을 모두 생성하는 요소를 예로 들 수 있겠다. 이러한 패드는 정적으로 또는 동적으로 생성이 가능하다.



요소를 시각적으로 표시하자면 다음 그림과 같다.

각 요소에는 고유의 상태가 있는데, 상태는 아래 값들 중 하나에 해당한다.

- **Null**: 요소의 기본 상태.
- **Ready**: 스트림을 사용할 준비가 되어 있고 흐르기를 기다리는 상태.
- **Paused**: 스트림은 열려 있는데 흐름이 freeze된 상태.
- **Playing**: 스트림이 열려 있고 흐르고 있는 상태.

파이프가 물을 시스템으로 전달하듯이 데이터는 버퍼를 이용해 시스템으로 전달된다. 전체적인 시스템은 제어 정보를 이동시키는 이벤트를 이용해 조직된다. 이러한 이벤트들은 요소로 전송되어, 전달된 이벤트에 따라 반응한다.

'이벤트' 혹은 GStreamer 용어를 이용하자면 '메시지'는 버스를 통해 이동한다. 버스는 파이프라인에 의해 생성된다. 그리고 우리는 버스를 이용해 그곳으로 발송된 메시지를 구독할 수 있다.

명령행을 이용해 GStreamer 파이프라인 접근하기

GStreamer는 명령행만을 이용해 파이프라인을 테스트하도록 도와주는 툴을 제공한다. 이는 파이프라인이 올바른지 여부를 빠르게 확인하는 데에 매우 유용하다. 구체적인 예를 들어보자. 스테레오 MP3 파일을 재생하길 원한다고 가정하자. 소스 요소는 MP3 파일 오프너(opener), 즉 filesrc가 되겠고, 스트림은 다음 요소인 mad, 즉 MP3 디코더로 전달한다. 이후 디코딩된 스트림은 raw 오디오 스트림을 가령 모노 채널 등으로 변환하는 audioconvert 요소로 전달된다.

이렇게 새로 수정된 스트림이 audioresample로 전달되면 8KHz의 오디오 스트림으로 변환한다. 그리고 마지막 스트림은 alsasink로 전달되어 스트림을 사운드 카드로 재생시킨다.

실행하기 - 파이프라인 테스트하기

이러한 파이프라인은 명령행과 GStream 툴인 gst-launch를 이용해 구현할 수 있다. 그 방법을 살펴보자.

1. Terminal을 열어 아래의 명령을 하나의 행으로 입력하라.

```
$ gst-launch-0.10 filesrc location=bass.mp3 ! mad ! audioconvert !
audio/x-raw-int,channels=1 ! audioresample ! audio/x-raw-int, rate=8000
 ! alsasink
```

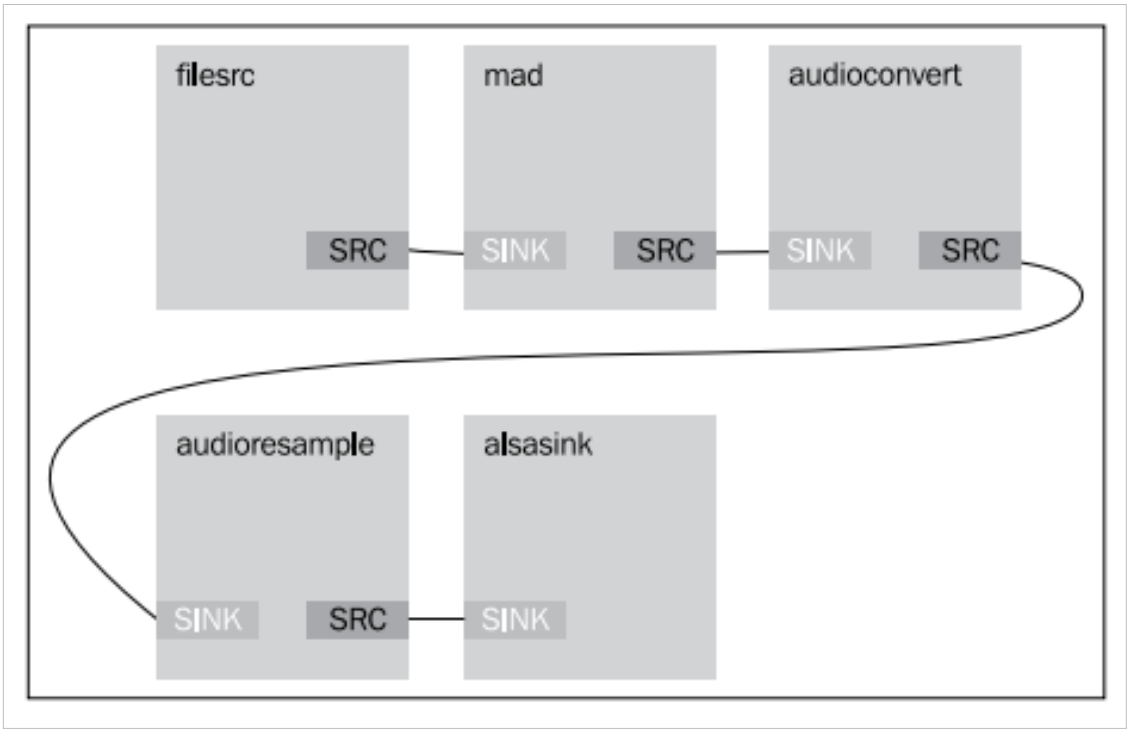
2. 오디오가 재생되는지 듣고 화면에 아래와 같은 내용이 출력되는지 확인하라.

```
Setting pipeline to PAUSED ...
Pipeline is PREROLLING ...
Pipeline is PREROLLED ...
Setting pipeline to PLAYING ...
New clock: GstAudioSinkClock
Got EOS from element "pipeline0".
```

```
Execution ended after 9311663370 ns.
Setting pipeline to PAUSED ...
Setting pipeline to READY ...
Setting pipeline to NULL ...
Freeing pipeline ...
```

무슨 일이 일어났는가**?**

파이프라인은 아래 그림과 같이 시각화할 수 있겠다.



gst-launch 툴은 명령행에 명시된 바와 같이 파이프라인을 구축한다. 우리는 요소들 사이에 느낌표를 이용해 서로를 연결한다.

가장 먼저 filesrc로 파이프라인을 시작하고, 재생하고자 하는 파일명을 입력함으로써 위치의 프로퍼티를 명시한다.

```
filesrc location=bass.mp3 !
```

이후 mad 라이브러리를 이용해 스트림을 mad 요소, 즉 MP3 디코더로 전달한다.

```
mad !
```

그 다음 스트림을 audioconvert 요소로 전달한다.

```
audioconvert ! audio/x-raw-int,channels=1 !
```

여기서 스트림을 채널이 하나인 정수 포맷의 오디오로 변환하기 위해 소스 패드를 명시한다. 그리고 audioresample 요소로 전달한다.

```
audioresample ! audio/x-raw-int, rate=8000 !
```

스트림을 8-KHz 오디오 스트림으로 변환하기 위해 소스 패드를 명시한다.

마지막으로 **"Advanced Linux Sound Architecture (ALSA)"**를 통해 스트림을 사운드 카드로 출력하는 alsasink 요소로 스트림을 전달한다.

```
alsasink
```

마지막으로 오디오가 재생되는 것을 듣는다!

실행하기 - 프로그램적으로 오디오 재생하기

명령행 툴을 이용해 GStreamer 요소와 상호작용하는 방법을 익히면 파이프라인 디자인이 작동할 것인지 여부를 확인하는 데에 매우 유용하다. 이를 프로그램적으로 실행해보도록 하겠다.

1. audio.js라는 새로운 스크립트를 생성하고 아래 코드로 채워라.

```
#!/usr/bin/env seed

GLib = imports.gi.GLib;
Gst = imports.gi.Gst;
GObject = imports.gi.GObject;

Main = new GType({
    parent: GObject.Object.type,
    name: "Main",
    init: function() {
        var pipeline = new Gst.Pipeline({ name: 'pipe' });
        var filesrc = Gst.ElementFactory.make ('filesrc', 'source');
        var mad = Gst.ElementFactory.make ('mad', 'decoder');
        var converter = Gst.ElementFactory.make ('audioconvert',
'converter');
        var resampler = Gst.ElementFactory.make ('audioresample',
'resampler');
        var alsasink = Gst.ElementFactory.make ('alsasink', 'sink');

        pipeline.add (filesrc);
        pipeline.add (mad);
        pipeline.add (converter);
        pipeline.add (resampler);
        pipeline.add (alsasink);

        filesrc.location = "bass.mp3";
        filesrc.link(mad);
        mad.link(converter);

        var caps = Gst.caps_from_string("audio/x-raw-int,channels=1")
converter.link_filtered(resampler, caps);
        caps = Gst.caps_from_string("audio/x-raw-int,rate=8000")
resampler.link_filtered(alsasink, caps);
        this.play = function() {
            pipeline.set_state (Gst.State.PLAYING);
```

```
        };
    }
});

Gst.init(Seed.argv);
var main = new Main();
main.play();

var context = GLib.main_context_default();
var loop = new GLib.MainLoop.c_new(context);

loop.run();
```

2. 아니면 UI와 라이센스 없이 audio.vala 라는 Vala 프로젝트를 생성할 수도 있다. src/audio.vala 파일을 아래 코드로 채워라.

```
using Gst;
using GLib;

public class Main : GLib.Object
{
    Pipeline pipeline;

    public Main ()
    {
        pipeline = Gst.parse_launch ("filesrc location=bass.mp3 !  mad
! audioconvert ! audio/x-raw-int,channels=1 ! audioresample !
audio/x-raw-int, rate=4000 ! alsasink") as Gst.Pipeline;
    }

    public void play() {
        pipeline.set_state (State.PLAYING);
        }

    static int main (string[] args)
    {
        Gst.init (ref args);
        var app = new Main ();
        app.play();

        new MainLoop().run();

        return 0;
    }
}
```

3. 아래 코드에서,

```
PKG_CHECK_MODULES(AUDIO, [gtk+-3.0 ])
```

configure.ac를 아래와 같이 수정하라.

```
PKG_CHECK_MODULES(AUDIO, [gstreamer-0.10 ])
```

4. 아래 코드에서,

```
audio_VALAFLAGS = \    --pkg gtk+-3.0
```

src/Makefile.am 을 아래와 같이 수정하라.

```
audio_VALAFLAGS = \
    --pkg gstreamer-0.10
```

5. 프로그램을 실행하는 디렉터리로 bass.mp3 파일을 복사할 것을 잊지 마라.
6. 프로그램을 실행하면 앞의 예제에서 재생된 것과 동일한 오디오가 들릴 것이다.

무슨 일이 일어났는가?

Vala와 JavaScript에서 작성된 코드 버전을 같이 살펴보면 둘 사이에 상당한 차이가 있음을 발견할 것이다. 이는 언어의 특성 때문이라기보다는 두 언어가 GStreamer와 상호작용하기 위해 대안적 방법을 사용하기 때문에 생긴 것이다.

JavaScript 코드를 먼저 살펴보자. 아래 코드에서 우리는 관련된 모든 요소들과 파이프라인을 정의한다.

```
var pipeline = new Gst.Pipeline({ name: 'pipe' });
var filesrc = Gst.ElementFactory.make ('filesrc', 'source');
var mad = Gst.ElementFactory.make ('mad', 'decoder');
var converter = Gst.ElementFactory.make ('audioconvert', 'converter');
var resampler = Gst.ElementFactory.make ('audioresample', 'resampler');
var alsasink = Gst.ElementFactory.make ('alsasink', 'sink');
```

각 요소에서 원하는 요소의 이름과 요소로 참조하길 원하는 이름을 선언해야 한다. 그 다음으로 접근 용이성을 위해 로컬 변수에 보관한다.

그리고 모든 요소를 파이프라인으로 추가한다.

```
pipeline.add (filesrc);
pipeline.add (mad);
pipeline.add (converter);
pipeline.add (resampler);
pipeline.add (alsasink);
```

이 때는 요소들이 추가되긴 했지만 서로 연결은 되지 않은 상태다. 따라서 아래와 같이 filesrc 요소의 location 프로퍼티를 명시한다.

```
filesrc.location = "bass.mp3";
filesrc.link(mad);
mad.link(converter);
```

이를 통해 bass.mp3 파일을 열고 싶다는 사실을 요소에게 알리는 셈이다. 또 filesrc 요소를 mad 요소로 연결한다. 그 다음으로는 mad 요소를 converter 요소로 연결함으로써 연결 작업을 계속한다.

아래의 코드는 converter 요소의 기능을 설정하는데, 가령 단일 채널을 이용하는 정수 포맷의 raw 오디오 스트림을 처리하는 기능을 들 수 있겠다.

```
var caps = Gst.caps_from_string("audio/x-raw-int,channels=1")
converter.link_filtered(resampler, caps);
```

이러한 기능들을 사용할 수 있다면, link 함수 대신 link_filtered 함수를 이용해 converter 요소를 resampler로 연결할 수 있다.

아래의 코드는 caps 변수를 재사용하고 그 값을 새 값으로 리셋한다.

```
caps = Gst.caps_from_string("audio/x-raw-int,rate=8000")
resampler.link_filtered(alsasink, caps);
```

앞의 코드에서 비트 전송률을 8000 Hertz로 설정한다. 이후 caps 변수가 있는 resampler 요소를 alsasink 요소로 연결한다. alsasink 요소는 사운드 카드에게 오디오를 재생시킬 것을 명령할 때 사용된다. 현재로선 더 이상 연결이 필요하지 않으므로 끝마친다.

오디오를 재생해야 할 경우 우리가 해야 할 일은 Gst.State.PLAYING 값으로 파이프라인의 상태를 트리거하는 것이다.

```
pipeline.set_state (Gst.State.PLAYING);
```

상태가 설정되면 흐름이 시작되어 사운드 카드에서 끝이 나면서 오디오를 들을 수 있게 된다.

Vala 버전에서는 이 결과를 도출하는 방법이 하나 이상임을 표시하기 위해 코드를 약간만 변경한다. 하지만 그보다 먼저 빌드 구조(build infrastructure)를 살펴보자.

gstreamer-0.10 pkgconfig 빌드 플래그를 우리의 빌드 구조로 포함시키도록 configure.ac를 수정함으로써 C 컴파일러가 헤더 파일을 비롯해 필요한 라이브러리를 어디서 찾을 것인지 인식하고 이해하도록 한다. 이 때 사용할 코드는 다음과 같다.

```
PKG_CHECK_MODULES(AUDIO, [gstreamer-0.10 ])
```

그리고 Gst 네임스페이스를 사용하길 원한다는 사실을 Vala 컴파일러가 알 수 있도록 Makefile.am 파일을 수정하는데, 해당 네임스페이스는 gstreamer-0.10 package에서 비롯된다.

```
audio_VALAFLAGS = \
    --pkg gstreamer-0.10
```

아래와 같이 한 행의 코드만으로 파이프라인을 구성할 수 있다.

```
pipeline = Gst.parse_launch ("filesrc location=bass.mp3 ! mad !
audioconvert ! audio/x-raw-int,channels=1 ! audioresample !
audio/x-raw-int, rate=4000 ! alsasink") as Gst.Pipeline;
```

앞의 코드에서는 명령행 코드 변환을 복사하여 Gst.parse_launch 함수로 붙여 넣을 뿐이다. 이 함수는 Gst.Element를 리턴하지만 우리는 캐스팅(casting)을 명시하기 위해 행 끝에 as Gst.Pipeline을 추가함으로써

Gst.Element를 Gst.Pipeline으로 변환해야 한다. 그 다음으로 상태를 설정함으로써 흐름을 시작한다.

```
pipeline.set_state (State.PLAYING);
```

두 가지 버전 모두에서 GLib 메인 루프를 사용하여 시스템이 이벤트에 관해 알고 즉시 종료하지 않도록 한다. 그 결과 프로그램을 종료하기 위해서는 Ctrl+C를 눌러야 하는데, 우리는 스트림의 끝을 알리는 이벤트는 처리하지 않기 때문이다.

실행하기 - 이벤트 처리하기

이제 각 이벤트에 적절하게 반응할 수 있도록 스트림으로부터 이벤트를 수신하는 법을 배워보자. 스트림이 끝나면 애플리케이션을 종료해야 한다고 가정해보자. 안타깝게도 이 때는 Vala를 이용할 수 밖에 없는데, Seed가 사용하는 GObject 자가점검 라이브러리의 현재 버전에는 우리의 목적을 달성하는 데 필요한 함수가 포함되어 있지 않기 때문이다. 그러한 이벤트를 처리하기 위해서는 아래 단계를 실행하라.

1. 오디오 프로젝트를 사용해 아래의 코드와 같은 모습이 되도록 audio.vala 파일을 수정하라.

```
public class Main : GLib.Object
{
    Pipeline pipeline;
    public signal void eos();

    bool bus_handler (Bus bus, Message message) {
        if (message.type == MessageType.EOS) {
            stdout.printf("End of stream!\n");
            eos();
        }
        return true;
    }

    public Main ()
    {
            pipeline = Gst.parse_launch ("filesrc location=bass.mp3 !
mad ! audioconvert ! audio/x-raw-int,channels=1 ! audioresample !
audio/x-raw-int, rate=4000 ! alsasink") as Gst.Pipeline;
        var bus = pipeline.get_bus ();
        bus.add_watch(bus_handler);
    }

    public void play() {
        pipeline.set_state (State.PLAYING);
        }

    static int main (string[] args)
    {
        Gst.init (ref args);
        var loop = new MainLoop();
        var app = new Main ();
        app.play();
```

```
        app.eos.connect(() => {
            loop.quit();
        });

        loop.run();

        return 0;
    }
}
```

2. 프로그램을 빌드하여 실행하라. 프로그램은 아래의 내용을 출력한 후 종료됨을 확인할 수 있을 것이다.

```
End of stream!
```

무슨 일이 일어났는가**?**

우리는 그저 스트림이 끝날 때마다 반응할 수 있길 원할 뿐이다. 따라서 이벤트를 수신할 때 콜백 함수를 호출하는 방법을 고려하고, 이벤트를 트리거하는 방법을 알아내면 되겠다.

아래의 코드는 파이프라인으로부터 GStreamer 버스를 얻는 데 사용할 수 있다.

```
var bus = pipeline.get_bus ();
bus.add_watch(bus_handler);
```

앞서 논의하였듯이 버스는 그 안으로 전송되는 모든 메시지를 이동시킨다(carry). 따라서 add_watch 함수를 이용해 버스를 감시해야 한다. 이후 다른 부분들을 준비해야 하는데, 먼저 우리만의 시그널을 설정한다.

```
public signal void eos();
```

두 번째는 시그널 핸들러가 되겠다.

```
bool bus_handler (Bus bus, Message message) {
    if (message.type == MessageType.EOS) {
        stdout.printf("End of stream!\n");
        eos();
    }
    return true;
}
```

앞의 코드에서는 우리만의 시그널을 준비하여 GStreamer 메시지를 우리 클래스의 클라이언트에게 이벤트로서 전달한다. 그 이유는 클라이언트가 GStreamer 메시지에 대해 전혀 알지 못할 수도 있기 때문에 GStreamer 이벤트를 우리만의 이벤트로 래핑(wrap)하여 시그널을 간소화하는 것이다.

메시지 핸들러는 꽤 간단한데, 메시지 타입이 **"End of Stream(EOS)"** 메시지인지 아닌지를 확인한다. 스트림 끝 메시지가 맞다면 텍스트를 출력하고 우리만의 시그널을 발생시키면 된다.

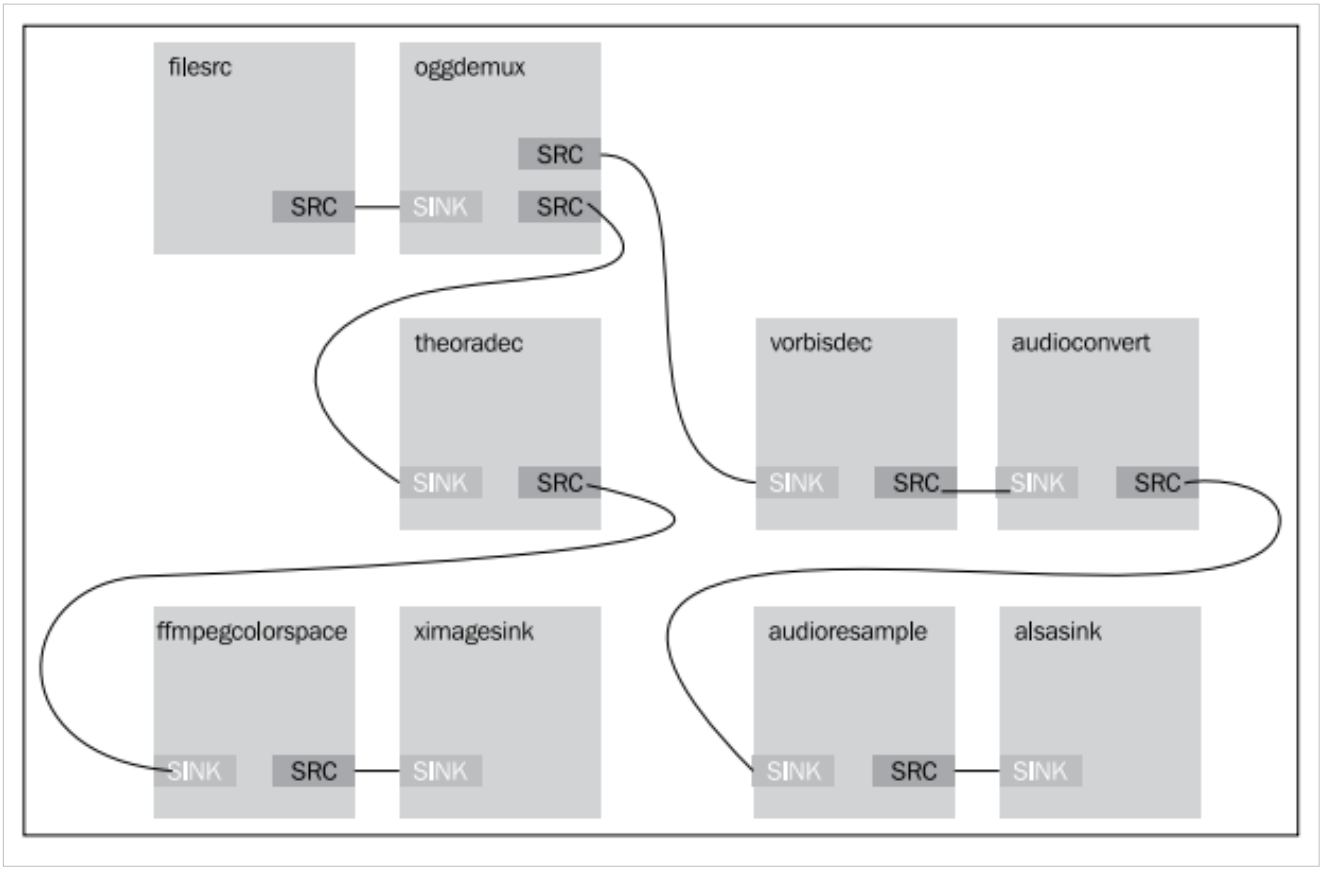
아래 코드는 클라이언트가 우리 시그널로 어떻게 연결되는지를 보여준다.

```
app.eos.connect(() => {
    loop.quit();
});
```

시그널을 수신하면 메인 루프를 종료하고 우리의 목적이 달성된다.

비디오 미디어 재생하기

이제 비디오를 재생해보자. 하지만 먼저 스트림 디자인을 생각해낼 필요가 있다. 아래 그림에 표시된 디자인이 우리의 요구를 충족시켜줄 것이다.



앞의 그림에서는 파일로부터 스트림이 **Ogg** 디멀티플렉서에 의해 두 부분으로 나뉨을 확인할 수 있다. 그 결과 소스 스트림은 오디오와 비디오 모두를 포함한다. 비디오 스트림은 **theora** 디코더와 색상 공간 변환기로 전달된다. 이러한 플러그인은 미디어로부터 비디오 내용을 디코딩하는 데 사용된다. 그리고 나면 **ximage** 싱크로 전달된다. 이는 디코딩된 자료를 화면에 표시하기 위함이다. 오디오 스트림은 **vorbis** 디코더로 전달된 후 오디오 변환기와 resampler 요소로 전달된다. **vorbis** 플러그인은 미디어에서 오디오 내용을 디코딩하는 데 사용된다. 이후 오디오를 사운드카드에서 재생하도록 **alsa** 싱크로 전달한다.

실행하기 - 비디오 재생하기

Wikipedia에서 받은 Al17_spill.org 파일을 재생할 것이다. 이 파일은 오디오가 있는 OggTheora 비디오 스트림이다. 아래 단계를 따라하여 비디오를 재생해보자.

1. 명령행 terminal에 아래 내용을 입력하라.

```
gst-launch-0.10 filesrc location="Ap17_spill.ogg" ! oggdemux name=demux
 demux. ! queue ! theoradec ! ffmpegcolorspace ! ximagesink demux. !
queue ! vorbisdec ! audioconvert !  audioresample ! alsasink
```

2. 아래 스크린샷과 같이 오디오와 비디오가 함께 재생되는 작은 창이 표시될 것이다.

무슨 일이 일어났는가**?**

명령행 코드는 앞서 표시한 디자인을 엄격하게 따른다. 여기서 Ap17_spill.org 파일을 연다.

```
filesrc location="Ap17_spill.ogg" !
```

이제 filesrc로 연결된 Ogg 디멀티플렉서를 생성하고, 디멀티플렉서를 demux로 명명한다.

```
oggdemux name=demux demux. !
```

여기서 Ogg 디멀티플렉서의 소스를 나눈다.

```
queue !
```

첫 번째 branch는 theora 디코더로 연결되어 theora 스트림으로부터 비디오 스트림을 얻는다.

```
theoradec !
```

그리고 ffmpecolorspace로 전달하여 화면으로 표시하는 데에 적합한 색상 공간을 얻는다.

```
ffmpegcolorspace !
```

마지막으로 비디오 스트림을 ximagesink로 전달하여 화면으로 표시한다.

```
ximagesink demux. !
```

demux라고 불리는 요소로 연결된 이 경로가 여기서 끝난다는 것을 나타내기 위해 demux 다음에 마침표를 넣는다.

```
queue !
```

여기서는 두 번째 branch에 또 다른 경로를 넣는다.

```
vorbisdec !
```

아래의 코드 행에서는 스트림을 이용해 오디오 스트림을 얻는다.

```
audioconvert ! audioresample ! alsasink
```

이러한 시퀀스는 오디오 스트림만 재생했던 앞의 예제와 비슷하다.

앞서 제시한 바와 같이 branch 이름이 마침표로 끝나도록 큐의 끝을 표시하는 한 명령행에 오디오나 비디오 스트림 중 무엇을 먼저 넣는지는 중요하지 않다.

시도해보기 - 오디오를 먼저 정의하기

한 번 시도해보는 건 어떨까? 파이프라인에 오디오를 먼저 정의하고 나서 명령행을 이용해 비디오를 정의해 보자. 결과에는 시각적으로나 음향적으로 차이가 없을 것이다.

실행하기 - 프로그램적으로 비디오 재생하기

이제 앞에서 제시한 스트림 흐름 디자인을 이용해 프로그램에서 구현하길 원한다고 치자. 프로그램의 UI는 아래에 **Play/Stop** 버튼이 배치된 매우 간단한 비디오 박스라고 가정하자. 아래 순서를 따라하라.

1. video 라는 새로운 Vala 프로젝트를 생성하라. 이번에는 GtkBuilder 기능을 사용해보자.
2. video.ui 파일을 편집하여 두 개의 항목이 있는 수직 박스를 넣어라. 상단 항목에는 DrawingArea 위젯을 넣고 하단 항목에는 버튼을 사용하라. DrawingArea 위젯이 확장 가능하도록 확보하라.
3. src/Makefile.am 이 아래의 내용을 포함하도록 수정하라.

```
video_VALAFLAGS = \
    --pkg gtk+-3.0 --pkg gstreamer-0.10 --pkg gstreamer-interfaces-0.10
 --pkg gdk-x11-3.0
```

4. configure.ac가 아래의 내용을 포함하도록 수정하라.

```
PKG_CHECK_MODULES(VIDEO, [gtk+-3.0 gstreamer-0.10
gstreamer-interfaces-0.10 gstreamer-plugins-base-0.10 gdk-x11-3.0])
```

5. 본 서적의 코드 번들에서 이용 가능한 video.vala 파일을 포함시켜라. 이 기능에서 주요 부분은 다음과 같다.

```
public class Main : GLib.Object
{
    const string UI_FILE = "src/video.ui";

    public Main ()
    {
        Builder builder;

        pipeline = new Pipeline ("video");
        src = ElementFactory.make ("filesrc", "filesrc");
        demux = ElementFactory.make ("oggdemux", "demux");
        queue1 = ElementFactory.make ("queue", "queue1");
        queue2 = ElementFactory.make ("queue", "queue2");
        theoraDecoder = ElementFactory.make ("theoradec", "theora");
        vorbisDecoder = ElementFactory.make ("vorbisdec", "vorbis");
```

```
        colorConverter = ElementFactory.make ("ffmpegcolorspace",
"colorspace");
        audioConverter = ElementFactory.make ("audioconvert", "audio");
        audioResampler = ElementFactory.make ("audioresample",
"resampler");
        audioSink = ElementFactory.make ("alsasink", "audiosink");
        videoSink = ElementFactory.make ("ximagesink", "videosink");

        pipeline.add_many (src, demux, theoraDecoder, vorbisDecoder,
queue1, queue2, colorConverter, audioConverter, audioResampler,
audioSink, videoSink);

        src.link (demux);

        demux.link_many(queue1, queue2);
        demux.pad_added.connect((element, src_pad) => {
            var caps = src_pad.get_caps();
            var name = caps.get_structure(0).get_name();

            Pad sink_pad = null;

            if (name == "video/x-theora") {
                    ink_pad = queue1.get_pad("sink");
            } else if (name == "audio/x-vorbis") {
                sink_pad = queue2.get_pad("sink");
            } else {
                return;
            }
            if (sink_pad != null && sink_pad.is_linked() == false) {
                src_pad.link (sink_pad);
            }
        });

        queue1.link_many (theoraDecoder, colorConverter, videoSink);
        queue2.link_many (vorbisDecoder, audioConverter,
audioResampler, audioSink);

        try
        {
            builder = new Builder ();
            builder.add_from_file (UI_FILE);
            builder.connect_signals (this);

            videoArea = builder.get_object ("drawingarea1") as Widget;
            videoArea.draw.connect(() => {
                var xoverlay = videoSink as XOverlay;
                var xid =
```

```
(ulong)Gdk.X11Window.get_xid(videoArea.get_window());
                overlay.set_xwindow_id(xid);
                return false;
            });

            var window = builder.get_object ("window") as Window;
            window.show_all ();
        }
        catch (Error e) {
            stderr.printf ("Could not load UI: %s\n", e.message);
        }

        var bus = pipeline.get_bus ();
        bus.add_signal_watch ();
        bus.message.connect((bus, message) => {

            if (message.type == Gst.MessageType.EOS) {
                stop();
                    pipeline.set_state (State.READY);
                    reopen();
            }
        });

        playButton = builder.get_object("button1") as Button;
        playButton.clicked.connect(() => {
            if (playing) {
                stop();
            } else {
                play();
            }
        });
        reopen ();
        stop ();
    }
}
```

6. 프로젝트 폴더에 Ap17_spill.org 파일이 있는지 확인하라.
7. 프로그램을 빌드하여 실행하라. 아래 스크린샷처럼 비디오를 재생 및 중지할 수 있고, 프로그램을 종료할 수도 있다.

무슨 일이 일어났는가**?**

이 연습문제에는 주의해서 살펴보아야 할 부분들이 몇 가지 있다.

첫 번째는 오디오와 비디오 스트림 소스를 큐의 다음 요소 내 싱크로 어떻게 연결하는지가 된다. 스트림이 재생되기 전에는 디멀티플렉서와 큐를 연결할 수 없다. 따라서 우리가 해야 할 일은 요소들을 디멀티플렉서 내에서 pad_added 시그널로 연결하는 일이다. 이번 예제에서는 스트림이 재생될 때마다 오디오 및 비디오 스트림 패드가 생성된다.

```
demux.pad_added.connect((element, src_pad) => {
    var caps = src_pad.get_caps();
    var name = caps.get_structure(0).get_name();

    pad sink_pad = null;
```

시그널 핸들러에서는 스트림이 비디오 스트림을 포함하는지(video/x-theora와 비교 확인) 확인한 후 패드를 queue1 요소와 연결해야 한다.

```
if (name == "video/x-theora") {
    sink_pad = queue1.get_pad("sink");
}
```

반대로 오디오 스트림을 얻으면 패드를 queue2와 연결해야 한다. 그 외의 경우는 아무 것도 실행하지 않는다. 그러면 아래 코드에서 볼 수 있듯이 audiosink와 videosink에 이르기까지 filesrc 요소로부터 모든 것이 연결되어 있을 것이다.

```
    else if (name == "audio/x-vorbis") {
        sink_pad = queue2.get_pad("sink");
    } else {
        return;
    }
    if (sink_pad != null && sink_pad.is_linked() == false) {
        src_pad.link (sink_pad);
    }
});
```

다음으로 중요한 것은 비디오 프레임을 재생하는 방식이다. ximagesink 싱크는 비디오를 표시하는 고유의 창을 갖고 있다. 비디오를 videoArea 위젯으로 부착하기 위해서는 draw 시그널을 연결해야 한다(아니면 위젯이 준비되었음을 알려주는, 즉 window 객체가 유효함을 알려주는 시그널이라면 어떤 것이든 괜찮다).

우리 핸들러는 videoSink를 XOverlay로 얻게 되는데, 이는 우리가 명시하는 창으로 비디오 프레임을 그리는 데 사용되는 특수 인터페이스다. 그리기를 실행하기 전에 먼저 그리고자 하는 창의 핸들 번호에 해당하는 xid 값을 찾아야 하는데, 이는 아래 코드를 이용하면 될 일이다.

```
videoArea.draw.connect(() => {
    var xoverlay = videoSink as XOverlay;
    var xid = (ulong)Gdk.X11Window.get_xid(videoArea.get_window());
    xoverlay.set_xwindow_id(xid);
    return false;
        });
```

이 창은 videoArea 위젯의 X11 창이다. 먼저 videoArea 위젯에 대해 Gdk.X11Window.get_xid 함수를 호출함으로써 xid 값을 얻는다. xid 값을 얻고 나면 xoverlay에게 프레임을 수신할 때마다 xid 값을 이용해 videoArea 위젯 안에서 렌더링할 것을 알린다.

다음은 스트림과 상호작용하는 방법을 살펴보자. 무언가를 재생할 때는 파이프라인의 상태를 PLAYING으로 설정하면 된다. 일시 정지를 원하면 PAUSE로 설정한다. 그리고 스트림 알림의 끝을 수신하거나 창을 닫을 때는 NULL로 설정하면 된다. NULL로 상태를 설정하고 나서 비디오의 재생을 가능하게 만들기 위해서는 filesrc 요소의 위치를 다시 설정할 필요가 있다.

요약

이번 장에서는 GStreamer를 이용해 오디오 및 비디오 파일을 재생하는 방법을 학습하였다. 또 GStreamer의 기본적인 내용을 논했다. GStreamer가 어떻게 작동하는지, 미디어를 재생하기 위해서는 GStreamer의 요소들을 어떻게 연결해야 하는지에 대해서도 학습했다. 명령행을 이용해 스트림 흐름 디자인을 만들고 나서 추후에 구현할 수 있다. 마지막으로 GTK+ 위젯과 함께 GStreamer를 사용하는 방법을 배웠다.

제 8장, 데이터 다루기에서는 데이터를 다루는 방법을 살펴볼 것이다. 데이터베이스의 데이터뿐만 아니라 다양한 소스로부터 데이터를 다루겠다.

Notes

GNOME3ApplicationDevelopmentBeginnersGuide:Chap 08

경고 : 문서를 렌더링할 수 없습니다 - 일반 텍스트로 출력합니다.

문제의 잠재적 원인은 다음과 같습니다: (a) PDF-작성기 소프트웨어의 버그 (b) 문제가 있는 미디어위키 문법 (c) 테이블이 너무 넓음

제 8 장 데이터 다루기 데이터 다루기데이터에 대해 이야기할 때는 데이터베이스뿐만 아니라 다른 소스로부터의 데이터까지를 지칭한다. 이는 데이터로 접근하고 조작하며 사용자에게 표시하는 것까지 포함한다. 데이터로 접근성이 좋다는 것은 더 나은 통합을 의미하는데, GNOME 은 이를 잘 구현한다. GNOME 은 해당 작업을 수행하도록 많은 API를 제공하는데, 이번 장에서는 이러한 API를 논하도록 하겠다. 본 장에서는 여러 소스로부터 데이터를 얻는 방법과 이것을 화면으로 표시하는 방법을 집중해서 살펴보고자 한다. 화면에 데이터를 표시하기 위해 GTK+ TreeView 위젯을 사용할 것이다. 또한 주소록으로부터 데이터를 수집할 수 있는 "Evolution Data Server(에볼루션 데이터 서버)" 라이브러리를 소개하고자 한다. 본 장의 내용을 간단하게 유지하기 위해 Seed와 Glade만 사용할 것이다. 본 장에서 학습하게 될 내용은 다음과 같다. TreeView를 이용해 데이터 표현하기 에볼루션 데이터 서버 아키텍처 에볼루션 데이터 서버로 주소록 접근하기 이제 데이터를 꺼내보자! TreeView를 이용해 데이터 표현하기GTK+ TreeView는 데이터의 트리 타입과 리스트 타입을 모두 표시하는 데 사용되는 위젯이다. 위젯의 디자인은 데이터 모델의 구현, 데이터를 어떻게 표현하는지, 데이터로 어떻게 접근하고 조작하는지를 논리적으로 구분하기 위해 모델-뷰-컨트롤러(MVC) 디자인을 사용한다. 위의 그림은 디자인 패턴을 시각적으로 설명한다. 그림을 설명하기 위해 Web에 있는 검색 엔진을 상상해보자. 최종 사용자든 코드의 조각이든 상관없이 사용자는 컨트롤러에서 무언가를 트리거한다. 실제로는 최종 사용자가 검색 쿼리를 Web으로 제출하는 코드, Search 버튼, 또는 컨트롤러가 무언가를 실행하도록 직접 트리거하는 무언가를 누를 때가 이 단계에 해당하겠다. 컨트롤러는 트리거에 따라 모델을 수정한다.우리 검색 엔진 예제에서는 서버상의 검색 엔진에 해당하겠다. 검색 엔진은 쿼리를 취하여 매개변수로서 형성한 후 모델에 입력으로 전달한다. 그러면 모델은 새로운 데이터를 생성한다. 새로운 데이터는 뷰 객체의 표현을 대체하고 새로고침한다. 이후 사용자는 이 새로운 표현을 확인할 수 있다. 구체적으로 말해, 표현은 현재 검색 결과를 표시하는 화면이 될 것이다. 이러한 접근법을 이용하면 동일한 모델과 컨트롤러의 구현을 가지면서 여러 유형의 표현을 가질 수 있다. 웹 브라우저에서 검색 엔진은 세부적인 결과를 표시하는 반면 스마트폰에서는 결과를 간단한 형태로 표시할 수 있는 것이다. GTK+에서 모델은 TreeModel 인터페이스에서 정의되는데, 이는 모델 공급자(provider)에 의해 구현되어야 한다. 하지만 GTK+는 즉시 사용이 가능한 간단한 모델을 두 개 제공하므로, 별도로 TreeModel 인터페이스를 구현하지 않아도 된다. 제공되는 모델들은 바로 ListStore와 TreeStore다. ListStore는 간단한 리스트 데이터 구조체에 사용되는 반면 TreeStore는 트리 데이터 구조체를 제공하는 데에 사용된다. 모델에서 우리는 유지하고자 하는 데이터의 열을 정의한다. 모델로 접근하고 싶다면 모델을 반복하기 위해 Iter 객체로부터 도움이 필요하다. Iter 객체는 데이터 모델에서 특정 레코드를 가리킨다. 데이터를 이동하면서 이 객체를 이동시킬 수 있다. 실행하기 - TreeView 사용하기 테이블에 데이터를 열거할 수 있는 간단한 애플리케이션을 만들길 원한다고 가정해보자. 테이블에서 데이터를 제거하기도 하고 새로운 데이터를 추가할 수도 있다면 좋겠다. 이는 아래 단계를 통해 구현할 수 있다. treeview.ui 라는 새로운 Glade UI 파일을 생성하라. 파일에 창을 삽입하라. 창에 항목이 두 개인 Box를 삽입하고 수직 박스로 만들어라. ScrollableWindow 창을 UI의 상단 부분에 위치시키고, 확장 가능하게 만들어라. 두 개의 항목을 포함할 수 있는 또 다른 Box를 하단 부분에 위치시키되, 이번에는 수평 박스로 만들어야 한다. 두 개의 버튼을 수직 박스의 각 열에 추가하고, 각각 Remove 와 Add 라벨로 된 btnRemove 와 btnAdd로 명명하라. TreeView 위젯을 (Control and Display에서 이용 가능) ScrollableWindow 창으로 추가하라. 모델을 요청하는 대화상자가 나타날 것인데, 타원형 버튼을 누르고 New 를 클릭하라. 모델 대화상자는 이제 자동으로 liststore1로 채워진다. Create 버튼을 클릭하라. 위젯 리스트에서 liststore1 을 선택하라.General 탭에서 선택된 리스트스토어를 store로 재명명하라. 그 아래 Columns 데이터에서 <define a new column> 을 클릭하면 gchararray 타입으로 된 텍스트 엔트리로 변환될 것이다. 이를 한 번 더 실행하여 gchararray 타입으로 된 엔트리가 총 두 개가 되도록 하라. treeview1 객체를 view로 재명명하라. view에서 treeselection1 객체를 찾아 selection으로 재명명하라. 여기까지 완료되면 우리의 Glade 파일은 아래 스크린샷과 같은 모습이 될 것이다. treeview.js 로 명명된 새로운 Seed 스크립트 파일을 생성하라. 파일을 아래 코드로 채워라. #!/usr/bin/env seed Gtk = imports.gi.Gtk; GObject = imports.gi.GObject; Main = new GType({ parent: GObject.Object.type, name: "Main", init: function(self) { var columns = { NAME: 0, ADDRESS: 1, } var ui = new Gtk.Builder() this.ui = ui; ui.add_from_file("treeview.ui"); var window = ui.get_object("window1"); window.resize(300, 400); window.show_all(); window.signal.destroy.connect(Gtk.main_quit); this.clients = {}; var view = ui.get_object("view"); var selection = ui.get_object("selection"); selection.signal.changed.connect(function(s) { var btnRemove = ui.get_object("btnRemove"); btnRemove.sensitive = true; }); var btnRemove = ui.get_object("btnRemove"); btnRemove.signal.clicked.connect(function() { var selection = view.get_selection(); if (selection) { var selected = {}; var valid = selection.get_selected(selected); if (valid && selected.iter) { var model = view.get_model(); model.remove(selected.iter); } } }); var btnAdd = ui.get_object("btnAdd"); btnAdd.signal.clicked.connect(function() { var selection = view.get_selection(); if (selection) { var selected = {}; var valid = selection.get_selected(selected); if (valid && selected.iter) { var model = view.get_model(); model.insert(selected.iter, 1); } } }); column = new Gtk.TreeViewColumn({title:'Name'}); cell = new Gtk.CellRendererText(); cell.editable = true; column.pack_start(cell); column.add_attribute(cell, 'text', columns.NAME); cell.signal.edited.connect(function(obj, path, text) { var store = view.get_model(); var path = new Gtk.TreePath.from_string(path); var iter = {}; store.get_iter(iter, path); store.set_value(iter.iter, columns.NAME, text); }); view.append_column(column); column = new Gtk.TreeViewColumn({title:'Address'}); cell = new

Gtk.CellRendererText(); cell.editable = true; column.pack_start(cell); column.add_attribute(cell, 'text', columns.ADDRESS); cell.signal.edited.connect(function(obj, path, text) { var store = view.get_model(); var path = new Gtk.TreePath.from_string(path); var iter = {}; store.get_iter(iter, path); store.set_value(iter.iter, columns.ADDRESS, text); }); view.append_column(column); var store = view.get_model(); var iter = {}; store.append(iter); store.set_value(iter.iter, columns.NAME, "Robert"); store.set_value(iter.iter, columns.ADDRESS, "North Pole"); } }); Gtk.init(Seed.argv); var main = new Main(); Gtk.main(); 애플리케이션을 실행하라. 필드를 클릭하면 데이터를 편집할 수 있다. 하나의 행을 선택하고 Add 를 클릭하면 새로운 행을 하나씩 추가할 수 있다. 선택된 행을 제거할 수도 있다. 출력된 내용은 아래의 스크린샷과 같다.무슨 일이 일어났는가?이번 연습문제에서는 TreeView가 어떻게 작동하는지를 학습하였다. TreeView를 뷰로 갖게 되었고, 두 개의 CellRendererText 위젯과 그에 연관된 TreeView 열도 생겼다. 이 모델에서는 ListStore를 사용한다. 먼저 유지하길 원하는 데이터를 정의해야 한다. 이는 두 개의 열로 된 데이터로, 각 열은 타입 문자열을 갖는다. 열 번호는 상수를 이용해 참조한다. 다음과 같이 열 번호 0에는 NAME을, 열 번호 1에는 ADDRESS를 사용한다. var columns = { NAME: 0, ADDRESS: 1, } 빠르게 접근할 수 있도록 TreeView 참조는 view 변수에 보관한다. var view = ui.get_object("view"); 선택된 내용의 변경된 시그널을 구독한다. 행이 선택될 때마다 이 코드가 호출될 것이다. 현재로선 유용한 내용이 포함되어 있지 않지만 실제 프로젝트에서는 특정 버튼을 활성화시키거나 알림을 표시하는 등의 일을 수행할 수 있다. 아래는 선택내용(selection) 코드에 해당한다. var selection = ui.get_object("selection"); selection.signal.changed.connect(function(s) { var btnRemove = ui.get_object("btnRemove"); btnRemove.sensitive = true; }); 아래 코드에서 레코드의 제거를 Remove 버튼의 클릭된 시그널로 연관시킨다. var btnRemove = ui.get_object("btnRemove"); btnRemove.signal.clicked.connect(function() { 먼저 아래 코드를 실행함으로써 현재 selection 객체를 얻는다. var selection = view.get_selection(); if (selection) { var selected = {}; var valid = selection.get_selected(selected); selection 객체가 유효해지면 선택된 행 객체를 얻는다. 레코드가 있다면 선택된 객체 안에 iter 객체가 얻어질 것이다. iter 객체를 얻을 때는 아래 코드를 이용해 레코드를 삭제할 수 있다. var model = view.get_model(); model.remove(selected.iter); 이후 Add 버튼에도 동일한 과정을 수행한다. 단, 이번에는 레코드를 제거하는 대신 새로운 레코 드를 삽입한다. var btnAdd = ui.get_object("btnAdd"); btnAdd.signal.clicked.connect(function() { var selection = view.get_selection(); if (selection) { var selected = {}; var valid = selection.get_selected(selected); if (valid && selected.iter) { var model = view.get_model(); model.insert(selected.iter, 1); } } }); 이제 열을 준비해보자. 각 열은 TreeViewColumn 위젯으로 표현된다. 각 TreeViewColumn 위젯은 CellRenderer 클래스를 이용해 연관시킨다. 우리 데이터는 일반 문자열이기 때문에 우리가 사용하는 렌더러는 CellRendererText가 된다. column = new Gtk.TreeViewColumn({title:'Name'}); cell = new Gtk.CellRendererText(); 다음으로 데이터를 열로 패킹한 다음 데이터를 편집할 수 있도록 셀을 편집 가능하게 만든다. cell.editable = true; column.pack_start(cell); add_attribute 함수를 이용해 ListStore에서 열, 셀, 데이터를 연관시킨다. 아래 코드는 열 번호 0(columns.NAME의 값)에서 얻은 데이터를 이용해 cell 객체의 text 프로퍼티를 수정할 것이다. 모든 것이 설정되고 나면 열을 view 변수 뒤에 추가한다. column.add_attribute(cell, 'text', columns.NAME); view.append_column(column); Anjuta에서 liststore1 위젯에 두 개의 gchararray 항목을 포함할 수 있도록 두 개의 열을 생성했던 일을 기억하는가? 이는 두 열 모두 gchararray (문자열에 대한 또 다른 이름) 타입임을 나타내기 위한 행동이었다. 첫 번째 열은 column.NAME 값을, 두 번째 열은 column.ADDRESS 값을 보유하기 위함이다. 다음으로 셀의 편집된 시그널을 연결한다. 핸들러에서 해야 할 일은 단순히 새로운 편집된(edited) 텍스트를 얻어 다시 우리 모델로 넣는 일이다. cell.signal.edited.connect(function(obj, path, text) { var store = view.get_model(); 먼저 인자에서 path 변수가 표시한 경로를 얻는다. ListStore가 해석할 수 있는 경로 표현을 얻는다. var path = new Gtk.TreePath.from_string(path); var iter = {}; 데이터를 설정하기 위해서는 현재 편집된 객체의 iter 객체를 얻을 필요가 있다. path 변수를 이용해 iter 객체로 변환한다. iter 객체를 얻은 후에는 set_value 함수를 이용해 모델을 설정한다. set_value에서는 편집하고자 하는 열을 명시하는 일도 해야 한다. store.get_iter(iter, path); store.set_value(iter.iter, columns.NAME, text); 주소에 대한 다른 열을 추가하는데, 이에 해당하는 코드는 이름 열에 사용된 코드와 동일하다. 초기 데이터를 모델로 추가한다. 그리고 하나의 행만 추가한다. 데이터를 추가하기 위해서는 먼저 모델을 얻을 필요가 있다. var store = view.get_model(); 다음으로 iter 객체를 모델 뒤에 추가한다. var iter = {}; store.append(iter); 이후 iter 객체를 이용해 값을 넣는다. store.set_value(iter.iter, columns.NAME, "Robert"); store.set_value(iter.iter, columns.ADDRESS, "North Pole"); 데이터를 얻고 조작하는 과정은 꽤 간단하게 보인다. 먼저 모델을 얻고 Iter 객체를 얻은 다음 상호작용하길 원하는 열을 명시함으로써 모델로부터 새 값을 얻거나 새 값을 모델로 추가하면 된다. 그 다음 가짜(fake) 데이터를 사용하는 대신 에볼루션 데이터 서버를 활용하여 라이브 데이터를 사용한다. 에볼루션 데이터 서버[EDS] 아키텍처EDS는 주소록, 달력, 작업(task)으로의 접근과 조작을 추상화한다. 여기에는 개발자가 특정 주소록, 달력 또는 작업 서비스로 접근하기 위한 플러그인을 작성할 수 있도록 해주는 플러그인 아키텍처가 있다. EDS 사용자는 추상화된 API를 사용하고, 문제가 있는 서비스의 세부 내용에 대해 알 필요가 없다. 이 접근법을 이용하면 GNOME은 이론적으로 어떤 유형의 주소록, 달력 또는 작업 서비스든 지원이 가능하다. 아래는 EDS 아키텍처의 구조를 간략하게 표시한 그림이다. EDS는 EDS의 사용자를 위해 메모리에서 유지되는 데몬을 제공하는데, 사용자는 보통 이메일 클라이언트, 인스턴스 메시징 애플리케이션, 또는 데이터로 접근해야 하는 애플리케이션에 해당한다. 서비스가 인증(authentication) 또는 승인(authorization)을 필요로 하는 경우 GNOME은 대화상자를 팝업시켜 최종 사용자가 가령 비밀번호를 채우는 일이나 접근을 허용 하는 일 등을 실행할 수 있도록 한다. EDS에서 주소록은 로컬 또는 원격 데이터 소스처럼 데이터의 소스를 나타내는 데이터 소스 그룹의 개념을 갖고 있다. 이것은 EBox.SourceGroup 객체에서 논리적으로 표현된다. 각 SourceGroup 그룹은 실제 데이터 소스를 표현하는 소스를 많이 포함할 수 있다. 이는 EBook.Source 객체에서 설명된다. 시도해보기 - 주소록과 달력 데이터 소스 준비하기 주소록과 달력 데이터로 접근하기 전에 먼저 데이터 소스를 준비할 필요가 있다. 이제 GNOME에서 Google 계정을 어떻게 설정하는지 논해보자. 시작하기 전에 활성화된 Google 계정을 갖고 있는지 확인하라. 이제 어떻게 계정을 준비시키는지 살펴보자. GNOME System Settings 를 열어라.Online Accounts 를 열어라. 창의 하단 좌측 모서리에 + 버튼을 클릭하라. GNOME과 사용하길 원하는 계정 유형을 선택하라. 이번 연습문제에는 Google services 를 선택한다. 서비스 로그인 페이지가 나타날 것이다. 서비스에 성공적으로 로그인하도록 하라. GNOME으로 접근을 허용하고, 해당 서비스를 사용하길 원함을 알려라. 모든 온라인 단계가 완료되면 컴퓨터에서 사용하길 원하는 서비스를 활성화하라. 특히 아래 스크린샷과 같이 Contacts 서비스를 활성화하라.무슨 일이 일어났는가?온라인 Google 계정의 연결이 끝났다. 즉, 이제 Google 계정에 GNOME APIs로 사용할 수 있는 데이터가 상주하고 있다는 말이다. 이것이 우리 다음 목표다. 실행하기 - 주소록 접근하기 간단한 주소록 프로그램을 생성하길 원한다고 가정하자. 데이터는 로컬 컴퓨터나 Google에서 원격으로 가져올 수 있다. 아래 단계를 실행하라. address-book.ui 라고 불리는 새로운 Glade .ui 파일을 생성하라. 두 개의 항목을 포함하는 수평 박스를 추가하라. 박스의 좌측면에 TreeView 위젯을 위치시켜라. bookView로 재명명하라. 모델을 요청하면 새로운 ListStore 모델을 생성하고 books로 재명명하라. TreeView 위젯에 자동으로 생성되는 TreeSelection 객체를 selection으로 재명명하라. ScrollableWindow 창을 박스

의 우측면에 놓아라. 이후 다른 TreeView를 ScrollableWindow 안에 넣어라. 이를 위한 또 다른 ListStore 모델을 생성하고 contacts로 재명명하라. TreeView를 contactView로 재명명하라. books로 명명된 ListStore 모델을 편집하라. 해당 모델 내부에 gchararray 타입으로 된 열을 두 개 추가하라. contact로 명명된 ListStore 모델을 편집하라. 다시 gchararracy 타입으로 된 두 개의 열을 모델 안에 추가하라. 이제 UI 디자인은 아래 스크린샷과 비슷한 모양이 될 것이다. address-book.js 라는 새로운 Seed 스크립트를 생성하라. 아래는 스크립트의 실행에 매우 중요한 역할을 하는 코드 블록이다. Main = new GType({ parent: GObject.Object.type, name: "Main", init: function(self) { var bookColumn = { UID: 0, NAME: 1, } var contactColumn = { NAME: 0, EMAIL: 1, } this.listContacts = function(e) { var c = {}; var q = EBook.BookQuery.any_field_contains(""); var r = e.get_contacts_sync(q.to_string(), c, null); if (r && c && c.contacts && c.contacts.length > 0) { var store = self.contact_view.get_model(); c.contacts.forEach(function(contact) { var iter = {}; store.append(iter); var name = contact.full_name; if (!name) { name = contact.nickname; } store.set_value(iter.iter, contactColumn.NAME, name); store.set_value(iter.iter, contactColumn.EMAIL, contact.email_1); }); } } this.clients = {}; var book_view = ui.get_object("bookView"); var selection = ui.get_object("selection"); selection.signal.changed.connect(function(s) { var selected = {} s.get_selected(selected); var book = selected.model.get_value(selected.iter, bookColumn.UID); var uid = book.value.get_string(); if (uid == "") { return; } source = self.sources.peek_source_by_uid(uid); var e = null; if (typeof(self.clients[uid]) !== "undefined") { e = self.clients[uid]; if (e) { self.clients[uid] = e; self.listContacts(e); } } else { var e = new EBook.BookClient.c_new(source); var r = e.open(false, null, function() { if (e) { self.clients[uid] = e; self.listContacts(e); } }); } }); var cell = new Gtk.CellRendererText(); var column = new Gtk.TreeViewColumn({title:'Book'}); column.pack_start(cell); column.add_attribute(cell, 'markup', bookColumn.NAME); book_view.append_column(column); var contact_view = ui.get_object("contactView"); this.contact_view = contact_view; cell = new Gtk.CellRendererText(); column = new Gtk.TreeViewColumn({title:'Name'}); column.pack_start(cell); column.add_attribute(cell, 'text', contactColumn.NAME); contact_view.append_column(column); cell = new Gtk.CellRendererText(); column = new Gtk.TreeViewColumn({title:'E-mail'}); column.pack_start(cell); column.add_attribute(cell, 'text', contactColumn.EMAIL); contact_view.append_column(column); var s = {}; var e = EBook.BookClient.get_sources(s); this.sources = s.sources; var groups = this.sources.peek_groups(); if (groups && groups.length > 0) { var store = book_view.get_model(); groups.forEach(function(item) { var iter = {}; store.append(iter); store.set_value(iter.iter, bookColumn.UID, ""); store.set_value(iter.iter, bookColumn.NAME, ""+item.peek_name()+ ""); var sources = item.peek_sources(); if (sources && sources.length > 0) { sources.forEach(function(source) { store.append(iter); store.set_value(iter.iter, bookColumn.UID, source.peek_uid()); store.set_value(iter.iter, bookColumn.NAME, source.peek_name()); }); } }); } } }); 코드를 실행하라. 애플리케이션이 실행되면 아래 스크린샷과 같은 창이 표시된다. 무슨 일이 일어났는가?앞의 연습문제 결과는 설정에 따라 모습이 다를 수도 있다. (텍스트를 흐리게 만든 것은 이메일 주소이기 때문에 개인 정보를 보호하기 위함이니 양해 바란다.) 위의 스크린샷에서는 EDS 가 On This Computer, On LDAP Servers, WebDAV, Google 이라는 네 개의 주소록 소스를 리턴한다. 이 소스들 중 실제 데이터를 포함하는 건 두 개, On This Computer 와 Google 밖에 없다. 스크린샷에 열거된 주소록의 이름은 Personal 과 Contacts 다.Contacts 를 클릭하면 모든 연락처의 리스트가 창의 우측면에 표시된다. 여기서는 이름과 이메일 주소, 두 개의 열만 표시한다.소스 코드를 자세히 살펴보자. 먼저 열에 대한 상수를 정의한다. 여기서 두 개의 모델을 갖고 있는데, 하나는 주소록 컬렉션(books라고 부른다)이고 하나는 연락처 컬렉션(contacts라고 부른다)용이다. 각 책에는 유일한 식별자의 이름이 있다. 따라서 열에 대해서는 아래의 데이터를 사용한다. var bookColumn = { UID: 0, NAME: 1, } 연락처의 경우, 이름과 이메일 주소만 표시하므로 두 개의 열만 있으면 된다. var contactColumn = { NAME: 0, EMAIL: 1, } bookView 변수의 참조는 .ui 파일에 보관하고 아래와 같은 코드 행에서는 book_view 변수 안에 저장한다. var book_view = ui.get_object("bookView"); bookView 변수의 selection 객체도 얻어 selection 변수에 넣는다. var selection = ui.get_object("selection"); 선택내용으로부터 책을 선택하면 해당하는 책의 내용을 얻길 원한다. 이를 구현하기 위해서는 아래와 같이 변경된 시그널을 함수로 접속(hook)해야 한다. selection.signal.changed.connect(function(s) { 핸들러에서는 먼저 선택내용으로부터 selected 객체를 얻는 일을 해야 한다. var selected = {} s.get_selected(selected); selected 객체에서는 iter member에 보관되는 Iter 객체를 얻는다. 우리는 즉시 열 번호 0의 값을 얻게 된다 (이 값은 bookColumn.UID를 이용해 기호화된다). 값은 타입 문자열에 있으므로 get_string() 함수를 이용한다. 우리는 uid 값이 비어 있을 때마다 행이 특정 책을 가리키지 않음을 의미하도록 행위를 설정하는 것이다. 이것은 주소록의 소스를 표시하기 위해 프로그램에 의해 사용 된 다. var book = selected.model.get_value(selected.iter, bookColumn.UID); var uid = book.value.get_string(); if (uid == "") { return; } uid가 어떤 값을 갖고 있다면 우리는 EDS에게 EBook.Source를 얻도록 직접 요청하는데, 이는 uid 값으로 식별된다. 이에 따라 e 변수에 저장된 EBook.Source 타입을 리턴할 것이다. 얻은 소스는 클라이언트 캐시에 보관하므로 책을 클릭할 때마다 매번 소스를 다시 열지 않아도 된다. 캐시가 uid 값으로 정의된 소스를 갖고 있다면 그냥 listContacts 함수를 호출하면 된다. 그러한 소스가 없다면 우선 열어 야 한다. 그리고 나서 캐시에 보관할 것이다. 이후 아래 코드 조각과 같이 listContacts를 이용해 내용을 열거한다. source = self.sources.peek_source_by_uid(uid); var e = null; if (typeof(self.clients[uid]) !== "undefined") { e = self.clients[uid]; if (e) { self.clients[uid] = e; self.listContacts(e); } } else { var e = new EBook.BookClient.c_new(source); var r = e.open(false, null, function() { if (e) { self.clients[uid] = e;

self.listContacts(e); } }); } }); 콜백을 함수의 인자로 제공함으로써 소스를 비동기식으로 연다는 점을 주목하라. 이 접근법을 이용하면 애플리케이션을 방해하지 않 고 사용자에게 비응답식(unresponsive)으로 만들어 소스를 열기 위한 시간을 제공할 수 있다. 소스가 만일 승인을 받아야 하고 우리의 주의를 끌어야 할 경우 이 시점에서 대화상자가 나타날 것이다. 여기서는 bookView를 위한 열 을 정의한다. 시각적으로는 하나의 열만 표시하는데, 책의 제목이나 책의 소스 그룹이 해당한다. var cell = new Gtk.CellRendererText(); var column = new Gtk.TreeViewColumn({title:'Book'}); column.pack_start(cell); 다음 코드 부분에 설명되어 있듯이 Pango 마크업을 사용한다. 그래야만 CellRendererText의 text 프로퍼티를 사용하는 대신 마크업을 이용해 열 1로 매핑할 수 있다 (bookColumn.NAME으로 표기). column.add_attribute(cell, ' markup', bookColumn.NAME); book_view.append_column(column); 아래 코드 조각에는 연락처에 해당하는 열을 정의한다. 눈에 보이는 열은 두 개가 있으며, 이들을 contactColumn.NAME과 contactColumn.EMAIL 이름으로 각각 할당할 것이다. var contact_view = ui.get_object("contactView"); this.contact_view = contact_view; cell = new Gtk.CellRendererText(); column = new Gtk.TreeViewColumn({title:'Name'}); column.pack_start(cell); column.add_attribute(cell, 'text', contactColumn.NAME); contact_view.append_column(column); cell = new Gtk.CellRendererText(); column = new Gtk.TreeViewColumn({title:'E-mail'}); column.pack_start(cell); column.add_attribute(cell, 'text', contactColumn.EMAIL); contact_view.append_column(column); 초기화 중에는 get_sources 함수를 이용해 주소록의 소스를 얻는다. 이후 시스템에서 이용 가능한 그룹은 peek_groups 함수를 이용해 찾는다. var s = {}; var e = EBook.BookClient.get_sources(s); this.sources = s.sources; var groups = this.sources.peek_groups(); if (groups && groups.length > 0) { var store = book_view.get_model(); 각 그룹에서 우리는 책 ID와 책 이름을 테이블로 추가하지만 uid 값은 채우지 않을 것인데, 실제 주소록 소스의 uid 값만 보관하길 원하기 때문이다. 그룹명은 아래 코드에서 볼 수 있듯이 볼드체와 이탤릭체 스타일의 마크업을 이용해 닫는다. groups.forEach(function(item) { var iter = {}; store.append(iter); store.set_value(iter.iter, bookColumn.UID, ""); store.set_value(iter.iter, bookColumn.NAME, "" +item.peek_name()+ ""); 앞의 코드 조각에서 HTML 마크업과 비슷한 것을 사용하였음을 주목하라. 사실 그들은 Pango 마크업으로, GNOME 프레임워크에서 사용되는 텍스트 렌더링 엔진인데 HTML과 비슷하지만 기능의 수는 비교적 적다. 여기에 마크업을 넣은 이유는 위젯을 표현 계층에서 디자인하 길 원하기 때문이다. 따라서 여기서 데이터를 건들거나 스타일을 추가해서는 안 되며, 위젯이 표시되면 스타일을 디자인하기만 한다. 하지만 앞의 접근법은 올바르지 않은데, 데이터를 저장공간에 넣기 전에 수정하기 때문이다. 데이터 내부를 검색하면 데이터 를 찾 을 수 없을 것인데, 이미 마크업을 이용해 채워졌기 때문이다. 사실상 렌더링 위젯 안에서 디자인하는 것이 올바른 구현일 것이다. 올바르게 구현하면 우리는 더 이상 Gtk.CellRendererWidget을 사용할 수 없게 되지만 대 신 데이 터를 표시하기 전에 디자인하는 커스텀 렌더러 위젯을 사용할 것이다. peek_sources 함수를 이용하면 각 그룹에 대한 실제 주소록 소스를 얻고, 이용 가능한 각 소스를 테이블로 넣을 것이다. 이제 열 0에 uid를 위치시키는데 (bookColumn.UID) 이는 아래 코드 조각에 표시하겠다. var sources = item.peek_sources(); if (sources && sources.length > 0) { sources.forEach(function(source) { store.append(iter); store.set_value(iter.iter, bookColumn.UID, source.peek_uid()); store.set_value(iter.iter, bookColumn.NAME, source.peek_name()); }); 열 0은 테이블에서 표시되지 않는데, 그곳에 TreeViewColumn을 추가하지 않은 까닭이다. listContacts 함수가 먼저 하는 일은 주소록으로 쿼리를 준비하는 일이다. EDS는 쿼리를 EDS로 전송하기 위해 EBook.BookQuery 객체를 제공한다. 아래 코드 조각을 통해 우리는 any_field_contains("") 함수를 이용해 쿼리를 생성함으로서 모든 데이터를 얻도록 EDS에게 요청할 것이다. this.listContacts = function(e) { var c = {}; var q = EBook.BookQuery.any_field_contains(""); var r = e.get_contacts_sync(q.to_string(), c, null); if (r && c && c.contacts && c.contacts.length > 0) { get_contacts_sync로 전달한 객체의 contacts member는 주소록의 연락처로 채워질 것이다. 우리가 얻은 연락처마다(EBook.Contact의 형태) 흥미로운 프로퍼티를 얻어 (full_name, 닉네임, email_1 프로퍼티만 원한다) 모델로 추가한다. var store = self.contact_view.get_model(); c.contacts.forEach(function(contact) { var iter = {}; store.append(iter); var name = contact.full_name; if (!name) { name = contact.nickname; } store.set_value(iter.iter, contactColumn.NAME, name); store.set_value(iter.iter, contactColumn.EMAIL, contact.email_1); }); } } 시도해보기 - 데이터를 주소록으로 저장하기 앞의 연습문제는 주소록의 내용만 표시하도록 구현하였다. 그렇다면 조금 더 개선하여 edit 버튼을 추가해보는 건 어떨까?이를 실행하기 위해 편집된 시그널을 얻을 때마다 새 텍스트가 EBook.Contact 구조체로 다시 저장되도록 확보하라. e.modify_contact_sync 함수를 호출할 수 있다면 그 뼈대(skeleton)는 다음과 같은 모습일 것이다. modifiedContact.full_name = newName; // and/or modifiedContact.email_1 = newEmail; // then save it with e.modify_contact_sync(modifiedContact, null); 소스를 다시 읽음으로써 모델을 새로고침해야 한다. 요약GNOME에서 데이터 프로그래밍의 중점은 Treeview 위젯 패밀리에 있다. 애플리케이션의 디자인은 TreeView 위젯의 MVC 디자인 패턴도 따라야 한다. 모델을 최신으로 유지하는 것 이 중요한데, 그래야만 사용자가 이미 무용지물인 렌더링된 데이터 대신 실제 데이터를 항상 눈으로 확인할 수 있다. 뷰는 주로 TreeViewColumn 객체와 CellRenderer 클래스로 표현된다. GTK+에서 사용할 준 비 가 된 CellRenderer 클래스는 많지만 하나의 커스텀 위젯이 필요하다면 언제든 인터페이스의 구현을 통해 새 것을 만들 수 있다. 이번 장에서는 EDS라는 데이터 소스를 하나 살펴보았다. 이는 주소록뿐 아니라 달력, 작업, 이메 일도 포함 한다. 이를 위해 에볼루션 프로그램이라는 중요한 애플리케이션을 생성하기도 했다. 다음으로는 HTML5 애플리케이션을 GNOME3와 통합이 가능하도록 만들고자 한다. 이를 어떻게 구현할 것인지는 다음 장에서 살펴보도록 하 자. Notes

GNOME3ApplicationDevelopmentBeginnersGuide:Chap 10

제 10 장 데스크톱 통합

데스크톱 통합

애플리케이션을 훌륭하게 만드는 한 가지 조건으로 플랫폼과의 정확한 통합을 들 수 있다. 통합이라 함은 모양과 느낌(look and feel)을 꾸미기 위해 기능을 재구현하는 대신 플랫폼으로부터 직접 기능을 얻음으로써 플랫폼으로 끊김 없이(seamlessly) 접근할 수 있음을 의미한다.

이번 장에서 플랫폼의 컨텍스트는 GNOME 데스크톱에 해당한다. 이는 GNOME 플랫폼에서 주요점에 해당하여 하나의 장을 통해 자세히 설명할 필요가 있다고 생각되었다. 본 장의 목표는 주 목표는 데스크톱 기능 중 일부, 이름하여 세션 관리, 런처, 키링(keyring), 알림 시스템을 활용하는 방법을 학습하는 데에 있다. 구체적으로 논하게 될 내용은 다음과 같다.

- GNOME 세션 관리와 대화하기
- 런처 설치
- 비밀 데이터를 GNOME Keyring에 저장하기
- 알림 시스템
- 통합을 정확하게 처리하는 툴인 D-Bus를 먼저 살펴보자.

D-Bus와 대화하기

D-Bus는 프로세스간 통신**(IPC)**과 원격 프로시저 호출**(RPC)** 시스템으로, freedesktop.org 데스크톱 구현에 의해 사용된다. GNOME은 해당 소프트웨어의 구현자들 중 하나에 해당한다. D-Bus는 애플리케이션들이 서로 대화할 수 있도록 만든다. D-Bus는 한 당사자가 버스에 명령이나 쿼리를 게시(post)하고 버스를 듣는 다른 당사자들은 요청된 명령에 액션을 취할 수 있는 버스 시스템을 사용한다. 여기에는 세 개의 구분된 채널이 있는데, 이는 "시스템 버스"(system bus), "세션 버스"(session bus), "프라이빗 버스"(private bus)에 해당한다.

시스템 버스는 시스템 전반적인 메시지용으로, 사용자 또는 하드웨어 알림의 생성을 예로 들 수 있다. 버스는 시스템에 의해 실행되고, 시스템에서 모든 실행자가 실행하는 D-Bus 인식 애플리케이션은 전부 버스를 듣고 응답할 수 있다. 두 번째 타입은 세션 버스로, 실행 중인 데스크톱의 사용자에 의해 실행된다. 동일한 세션 내에 동일한 사용자가 실행하는 애플리케이션은 모두 버스를 듣고 그에 응답할 수 있다. 세 번째는 프라이빗 버스로, 점대점(point-to-point) 버스이며, 연결된 당사자들만 대화할 수 있다.

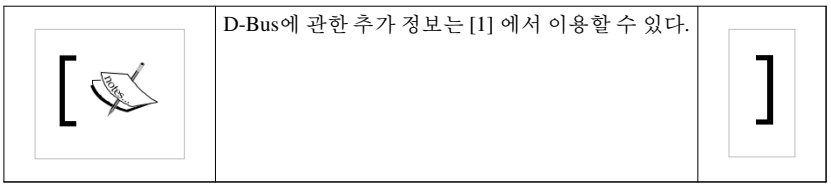
각 애플리케이션은 버스로 연결을 구축할 수 있다. 각 연결의 이름은 인터넷 도메인명을 역순으로 열거한 모습으로, org.gnome.SettingsDaemon.Power 를 예로 들 수 있겠다. 애플리케이션은 이후 경로라고 불리는 식별자를 이용해 버스에서 일부 서비스를 노출시킬 수 있다. 경로는 파일시스템 경로처럼 생겼는데, /org/gnome/SettingsDaemon/Power 를 예로 들 수 있다.

애플리케이션은 org.freedesktop 인터페이스와 같은 공통 인터페이스에 따라 서비스를 노출하거나, 고유의 인터페이스를 제공할 수 있다. 인터페이스는 명세(specification)에 따라 인터페이스의 사용자와 게시자(publisher)에 의해 적절하게 구현되어야 하는 API와 그 모습이 정확히 똑같다.

구체적으로 말해, 애플리케이션은 수신자의 경로와 연결 이름을 명시함으로써 명령을 포함한 메시지를 버스로 게시(post)할 수 있다. 연결명을 소유하고 명시된 경로를 노출하는 애플리케이션은 필요할 경우 인터페이스에 따라 다른 메시지로 반응 및 응답한다.

그 외의 경우, 애플리케이션은 명시된 경로를 이용해 시그널이 발생하였다는 메시지를 방송할 수도 있다. 시그널에 관심이 있는 또 다른 애플리케이션은 버스를 도청하여 시그널의 발생을 듣고 시그널을 수신 시 어떠한 실행을 할 수 있다.

D-Bus는 GNOME 데스크톱과 꽤 훌륭한 통합을 확보하는 데에 중요한 툴들 중 하나다. 플랫폼에서 이용 가능한 많은 기능들은 D-Bus를 이용해 접근할 수 있다. D-Bus가 어떻게 GNOME 데스크톱과의 통합을 돕는지에 대한 통찰력을 얻기 위해 D-Bus 세션을 들어보자.



실행하기 - D-Bus 듣기

D-Bus 세션 버스에서 무슨 일이 일어나는지 확인하기 위해 아래와 같은 실험을 해보자.

1. Terminal을 열어라.
2. dbus-monitor를 입력하고 Enter를 눌러라.
3. 화면은 D-Bus에서 온 메시지로 가득찰 것이다. 음향 크기를 증가시키거나 GNOME 메인 메뉴로 접근하거나 애플리케이션을 활성화하라. 당신의 액션은 모두 버스로 방송될 것이다.

무슨 일이 일어났는가**?**

dbus-monitor 명령은 세션 버스에 무엇이 공시되든지 내용을 듣는다. 랩탑 컴퓨터를 충전하면서 사용 중이라면 아래와 같은 내용이 뜰 것이다.

```
signal sender=:1.6 -> dest=(null destination) serial=192
path=/org/gnome/SettingsDaemon/Power;
interface=org.freedesktop.DBus.Properties;
member=PropertiesChanged
    string "org.gnome.SettingsDaemon.Power"
    array [
        dict entry(
            string "Tooltip"
            variant string "Laptop battery 1 hour 10 minutes until
charged (44%)"
        )
    ]
    array [
    ]
```

출력 내용은 org.freedesktop.DBus.Properties 인터페이스에 PropertiesChanged 시그널을 표시하고 있다고 말한다. 인터페이스에 따르면 시그널에는 세 개의 인자, 즉 문자열, 배열, 또 다른 배열이 있다. 이러한 특정 시그널은 /org/gnome/SettingsDaemon/Power 경로에서 노출되고 모두에게 방송된다. 첫 번째 인자 org/gnome.SettingsDaemon.Power, 사전(dictionary) 기록을 포함하는 배열, 빈 배열을 전송한다.

시스템 버스에서 무슨 일이 일어나는지 관심이 있다면 시스템 인자를 제공하라.

팝 퀴즈 - 훌륭한 애플리케이션 예제?

Q1. 앞의 데이터가 버스에 항상 게시될 경우 다음 중 데이터를 소모하기에 가장 적절한 애플리케이션 예는 무엇인가?

1. 배터리 시스템 트레이 애플릿(applet)
2. 배터리 확인 애플리케이션

GNOME 세션 관리자

GNOME 세션 관리자는 사용자의 데스크톱 환경 세션을 책임진다. 이는 시작 애플리케이션과 데스크톱 셸을 실행하고, 사용자의 로그아웃을 가능하게 하며, 컴퓨터를 종료한다. 애플리케이션은 심지어 관리자에게 로그아웃이나 종료를 요청할 수도 있다. 예를 들어, 사용자가 로그아웃을 원하는데 애플리케이션에 저장하지 않은 문서가 있다면 사용자가 문서를 저장하거나 명시적으로 애플리케이션을 종료할 때까지 로그아웃은 임시적으로 중지될 것이다.

세션은 사용자의 존재를 추적하고, 사용자가 일반 상태인지(available), 다른 작업 중인지(busy), 유휴 상태인지(idle), 보이지 않는 상태인지(invisible)도 추적한다. 사용자는 존재의 상태를 텍스트로 설정할 수도 있다.

실행하기 - 세션 관리자에게 이야기하기

1. session.js라고 불리는 스크립트를 생성하라 (소스 코드 배포판에서 이용 가능하다). 중요한 부분을 아래에 표시하겠다.

```
var SessionManagerInterface = {
    name: "org.gnome.SessionManager",
    methods: [
        { name: 'CanShutdown', inSignature: '', outSignature: 'b' },
        { name: 'Logout', inSignature: 'u', outSignature: '' },
        { name: 'Shutdown', inSignature: '', outSignature: '' },
        { name: 'Inhibit', inSignature: 'susu', outSignature: 'u' },
        { name: 'Uninhibit', inSignature: 'u', outSignature: '' }
    ]
}

Presence.prototype = {
    _init: function() {
        DBus.session.proxifyObject(this,
            'org.gnome.SessionManager',
            '/org/gnome/SessionManager/Presence');
    }
}

var PresenceInterface = {
    name: "org.gnome.SessionManager.Presence",
    methods: [
        { name: 'SetStatus', inSignature: 'u', outSignature: '' },
        { name: 'SetStatusText', inSignature: 's', outSignature: '' },
    ]
}
Main = new GType({
    parent: GObject.Object.type,
    name: "Main",
    init: function(self) {
        DBus.proxifyPrototype(SessionManager.prototype,
SessionManagerInterface);
        DBus.proxifyPrototype(Presence.prototype, PresenceInterface);
        this.manager = new SessionManager();
        this.presence = new Presence();
```

```
        ...

        var combo = ui.get_object("presenceStatus");
        cell = new Gtk.CellRendererText();
        combo.pack_start(cell);
        combo.add_attribute(cell, "text", 1);

        combo.signal.changed.connect(function(s) {
            var selected = {}
            s.get_active_iter(selected);
            var id = s.model.get_value(selected.iter, 0);
            self.presence.SetStatusRemote(id.value.get_int());
        });

        var textStatus = ui.get_object("textStatus");
        textStatus.signal.changed.connect(function(b) {
            self.presence.SetStatusTextRemote(textStatus.text);
        });

        var logout = ui.get_object("logOut");
        logout.signal.clicked.connect(function(b) {
            self.manager.LogoutRemoteSync(0);
        });

        var shutdown = ui.get_object("powerOff");
        shutdown.signal.clicked.connect(function(b) {
            self.manager.ShutdownRemoteSync();
        });

        var inhibit = ui.get_object("inhibit");
        inhibit.signal.toggled.connect(function(b) {
            if (inhibit.active == 1) {
                inhibit.label = "Uninhibit";
                var window = ui.get_object("window1");
                var xid = window.get_window().get_xid();
                self.inhibitCookie =
self.manager.InhibitRemoteSync(applic ationId, xid, "I forbid you to
logout", 1);
            } else {
                self.manager.UninhibitRemoteSync(self.inhibitCookie);
                inhibit.label = "Inhibit";
            }
        });
        window.show_all();
    }
});
```

2. 아니면 GtkBuilder를 지원하는 Vala 프로젝트를 생성하여 session-vala로 명명하라.
3. src/Makefile.am에서 아래와 같은 행을 찾아라.

```
session_vala_VALAFLAGS = \
    --pkg gtk+-3.0 --pkg gdk-x11-3.0
```

4. 위의 행을 아래와 같이 편집하라.

```
session_vala_VALAFLAGS = \
    --pkg gtk+-3.0
```

5. 아래 코드와 같이 가장 중요한 부분이 포함된 src/session_vala.vala를 생성하라.

```
using GLib;
using Gtk;

[DBus (name = "org.gnome.SessionManager")]
interface SessionManager : GLib.Object {
    public abstract bool can_shutdown () throws IOError;
    public abstract void logout (uint32 mode) throws IOError;
    public abstract void shutdown () throws IOError;
    public abstract uint32 inhibit (string appId, uint32 xid, string
reason, uint32 flags) throws IOError;
    public abstract void uninhibit (uint32 cookie) throws IOError;
}
[DBus (name = "org.gnome.SessionManager.Presence")]
interface Presence: GLib.Object {
    public abstract void set_status_text (string text) throws IOError;
    public abstract void set_status (uint32 mode) throws IOError;
}
public class Main : Object
{
    ...
    public Main ()
    {
            manager = Bus.get_proxy_sync(BusType.SESSION,
"org.gnome.SessionManager", "/org/gnome/SessionManager");
            presence = Bus.get_proxy_sync(BusType.SESSION,
"org.gnome.SessionManager",
"/org/gnome/SessionManager/Presence");
        ...
        var combo = builder.get_object("presenceStatus") as ComboBox;
        var cell = new CellRendererText();
        combo.pack_start(cell, true);
        combo.add_attribute(cell, "text", 1);

        combo.changed.connect((object) => {
            TreeIter iter;
```

```
            object.get_active_iter(out iter);
            Value value;
            object.model.get_value(iter, 0, out value);
            presence.set_status(value.get_int());
        });

        var textStatus = builder.get_object("textStatus") as Entry;
        textStatus.changed.connect(() => {
            presence.set_status_text(textStatus.text);
        });

        var logout = builder.get_object("logOut") as Button;
        logout.clicked.connect(() => {
            manager.logout(0);
        });

        var shutdown = builder.get_object("powerOff") as Button;
        shutdown.clicked.connect(() => {
            manager.shutdown();
        });

        var inhibit = builder.get_object("inhibit") as ToggleButton;
        inhibit.toggled.connect((object) => {
            if (object.active == true) {
                object.label = "Uninhibit";
                var window = builder.get_object("window1") as Window;
                var xid = Gdk.X11Window.get_xid(window.get_window());
                cookie = manager.inhibit("MyApplication", (uint32) xid,
 "I forbid you to logout", 1);
            } else {
                manager.uninhibit(cookie);
                object.label = "Inhibit";
            }
        });
    }
}
```

6. Glade로 UI를 생성하고 session.ui라고 불러라.
7. **Entry**, **ComboBox**, 버튼 집합을 넣어라. 첫 번째 버튼은 **ToggleButton** 이고 나머지는 일반 버튼이다.
8. **Entry** 에 textStatus를, **ComboBox** 에는 presenceStatus를, **ToggleButton** 에는 inhibit을, 나머지 **Button** 객체에는 **logOut** 과 **powrOff** 라는 이름을 부여하라.
9. **ComboBox** 에서 타원형 버튼을 클릭하여 새로운 모델을 생성하고, 이를 liststore1이라 불러라.
10. 제 8장 데이터 다루기에서 데이터 소스와의 상호작용 시 학습한 단계를 따라 두 개의 필드를 liststore1 로 넣어라. 첫 번째 데이터는 **gint** 로 표시되는 정수이고 두 번째는 **gchararray** 문자열이다.
11. liststore1을 아래의 데이터 쌍으로 미리 채워라.

```
0 Available
1 Invisible
```

```
2 Busy
3 Idle
```

12. **presenceStatus** 위젯으로 돌아가, **Active** 항목을 0 값으로 설정하라.
13. UI를 저장하고 애플리케이션을 실행하라. 아래 스크린샷과 같은 모습이 보일 것이다.

14. 텍스트를 텍스트박스로 입력함으로써 상태 텍스트를 넣을 수도 있다. 콤보박스에서 텍스트를 선택함으로써 존재의 상태를 변경할 수도 있다. **Log Out** 버튼을 누르면 로그아웃을 동의하는 즉시 세션이 닫히며, **Power Off** 버튼 또한 동일하게 작용하는데, 차이점이 있다면 **Power Off** 는 컴퓨터를 종료한다는 점이다. Inhibit 버튼을 활성화하면 로그아웃과 종료 액션이 취소될 것이다.

무슨 일이 일어났는가**?**

우리는 D-Bus API를 이용해 GNOME 세션 관리자와 상호작용을 하였다. 첫 번째 단계는 프록시 객체를 생성하는 것이다. 이 객체는 D-Bus API와 이어주는 다리 역할을 한다. 따라서 Seed에서는 접근할 수 없는 실제 D-Bus API를 호출하는 대신 프록시를 통해 호출하는 것이다.

프록시를 준비하기 위해서는 프로토타입을 이용해 JavaScript 객체를 생성하기만 하면 된다. 객체에서 초기화 함수를 정의하면 이 함수는 DBus.session.proxifyObject 함수를 호출한다. 이 함수는 JavaScript 함수와 D-Bus 함수 호출을 연결한다.

이 실험에서 우리는 세션 관리자로부터 두 개의 API 집합으로 접근한다. 첫 번째 집합은 SessionManager이고, 두 번째는 Presence다. 즉, 애플리케이션에 두 개의 프록시 객체가 설치되어 있을 것이란 의미다.

첫 번째 프록시는 연결명, 즉 org.gnome.SessionManager, 그리고 함수가 정의 및 노출된 경로, 즉 /org/gnome/SessionManager 를 명시하여 생성된다.

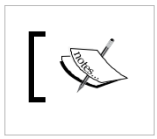
```
function SessionManager() {
    this._init();
}

SessionManager.prototype = {
    _init: function() {
        DBus.session.proxifyObject(this,
            'org.gnome.SessionManager',
            '/org/gnome/SessionManager');
    }
}
```

이후 이 객체로 매핑하길 원하는 인터페이스를 구현한다. 인터페이스는 접근하고자 하는 D-Bus 세계로부터 모든 메서드, 시그널, 프로퍼티를 열거한다. 이번 예제에서는 목적을 달성하기 위해 연결명과 메서드만 정의한다. 메서드 설명에는 inSignature와 outSignature가 있음을 주목하라. inSignature는 우리가 함수로 전달하는

매개변수를 지칭하고, outSignature는 함수가 리턴하는 변수를 나타낸다.

```
var SessionManagerInterface = {
    name: "org.gnome.SessionManager",
    methods: [
        { name: 'CanShutdown', inSignature: '', outSignature: 'b' },
        { name: 'Logout', inSignature: 'u', outSignature: '' },
        { name: 'Shutdown', inSignature: '', outSignature: '' },
        { name: 'Inhibit', inSignature: 'susu', outSignature: 'u' },
        { name: 'Uninhibit', inSignature: 'u', outSignature: '' }
    ]
}
```

앞의 설명을 바탕으로 다섯 개의 함수, CanShutdown, Logout, Shutdown, Inhibit, Uninhibit를 갖는다. 이 함수들은 org.gnome.SessionManager 연결에서 정의된다.



연결에서 노출되는 함수라고 해서 모두 우리 프록시에서 정의되어야 할 필요는 없다. 사용하고자 하는 함수만 정의하면 된다.

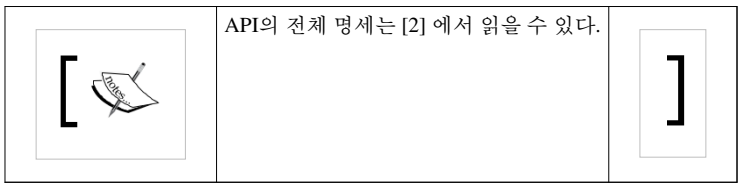
매개변수는 단일 문자열로 전달되는데, 각 문자는 D-Bus 규칙에 따른 변수 타입을 나타낸다. 아래 표는 D-Bus 명세 버전 0.19에서 복사한 타입을 열거한다.

일반적 이름	코드	설명
INVALID	0 (ASCII NUL)	유효하지 않은 타입 코드로, 시그니처를 종료하는 데에 사용되었다.
BYTE	121 (ASCII "y")	8-비트의 부호가 없는 정수.
BOOLEAN	98 (ASCII "b")	Boolean값으로, 0은 FALSE를 의미하고 1은 TRUE를 의미한다. 나머지는 모두 유효하지 않은 값이다.
INT16	110 (ASCII "n")	16-비트의 부호가 있는 정수.
UINT16	113 (ASCII "q")	16-비트의 부호가 없는 정수.
INT32	105 (ASCII "i")	32-비트의 부호가 있는 정수.
UINT32	117 (ASCII "u")	32-비트의 부호가 없는 정수.
INT64	120 (ASCII "x")	64-비트의 부호가 있는 정수.
UINT64	116 (ASCII "t")	64-비트의 부호가 없는 정수.
DOUBLE	100 (ASCII "d")	IEEE 754 double.
STRING	115 (ASCII "s")	UTF-8 문자열(유효한 UTF-8이어야 한다). null로 끝나야 하고, 다른 null 바이트는 포함해선 안 된다.
OBJECT_PATH	111 (ASCII "o")	객체 인스턴스명.
SIGNATURE	103 (ASCII "g")	타입 시그니처.

ARRAY	97 (ASCII "a")	배열.
STRUCT	114 (ASCII "r"), 40 (ASCII "("), 41 (ASCII ")")	구조체, 타입 코드 114 "r"은 바인딩과 구현에서 구조체의 일반적 개념을 표현할 때 사용되도록 예약(reserved)되며, D-Bus에서 사용되는 시그니처에 표시되어선 안 된다.
VARIANT	118 (ASCII "v")	Variant 타입(값의 타입이 값 자체의 일부).
DICT_ENTRY	101 (ASCII "e"), 123 (ASCII "{"), 125 (ASCII "}")	dict 또는 map의 엔트리 (키-값 쌍). 타입 코드 101 "e"는 바인딩과 구현에서 dict 또는 dict 엔트리의 일반적 개념을 표현하기 위해 사용되도록 예약되며, D-Bus에서 사용되는 시그니처에 표시되어선 안 된다.
UNIX_FD	104 (ASCII "h")	Unix 파일 기술자.
(reserved)	109 (ASCII "m")	GVariant의 것과 호환되는 "maybe" 타입을 위해 예약되며, 이곳에서 명시되기 전에는 D-Bus에서 사용되는 시그니처에 표시되어선 안 된다.
(reserved)	42 (ASCII "*")	바인딩과 구현에서 하나의 완전한 타입을 표현하기 위해 사용되도록 예약되며, D-Bus에서 사용되는 시그니처에 표시되어선 안 된다.
(reserved)	63 (ASCII "?")	바인딩과 구현에서 기본 타입을 표현하기 위해 사용되도록 예약되며, D-Bus에서 사용되는 시그니처에 표시되어선 안 된다.
(reserved)	64 (ASCII "@"), 38 (ASCII "&"), 94 (ASCII "^")	바인딩과 구현에서 내부적 사용을 위해 예약되며, D-Bus에서 사용되는 시그니처에 표시되어선 안 된다. GVariant는 이러한 타입 코드를 이용해 호출 규칙을 인코딩한다.

두 번째 프록시 객체는 Presence API를 위한 객체다. API는 텍스트와 수치 상태를 모두 설정하는 데에 사용된다. 수치 상태는 아래의 상수와 같다.

- 0 은 이용 가능한 상태
- 1 은 보이지 않는(invisible) 상태
- 2 는 바쁨(busy) 상태
- 3 은 유휴(idle) 상태



API는 org.gnome.SessionManager.Presence 연결에서 정의된다.

```
Presence.prototype = {
    _init: function() {
        DBus.session.proxifyObject(this,
            'org.gnome.SessionManager',
            '/org/gnome/SessionManager/Presence');
    }
}
```

인터페이스는 두 개의 함수만 필요하기 때문에 더 짧은데, 하나는 수치 상태용 SetStatus이고, 나머지 하나는 텍스트 상태용 SetStatusText이다.

```
var PresenceInterface = {
    name: "org.gnome.SessionManager.Presence",
```

```
    methods: [
        { name: 'SetStatus', inSignature: 'u', outSignature: '' },
        { name: 'SetStatusText', inSignature: 's', outSignature: '' },
    ]
}
```

프록시 객체로 함수를 매핑한 후에는 보통처럼 함수를 호출할 수 없다. Seed는 셋업 과정에서 인터페이스에 정의한 모든 함수 이름 뒤에 Remote와 RemoteSync를 추가한다. 따라서 애플리케이션은 뒤에 Remote 또는 RemoteSync가 붙은 함수명을 호출한다. RemoveSync 버전은 동기식 호출인 반면 Remote 호출은 함수와 비동기식으로 이루어진다.

셋업 과정은 아래와 같이 proxifyPrototype 함수를 호출하면 실행된다.

```
DBus.proxifyPrototype(SessionManager.prototype,
SessionManagerInterface);
DBus.proxifyPrototype(Presence.prototype, PresenceInterface);
```

즉, 인터페이스 객체에 정의된 인터페이스를 바탕으로 Seed가 Remote와 RemoteSync 함수 호출을 프로토타입에서 채울 것이란 의미다. 제 3장, 프로그래밍 언어에서 학습하였듯이 JavaScript에서는 함수를 객체의 member로 추가할 수 있다.

다음은 일부 변수를 새로 생성된 프록시 객체의 인스턴스가 되도록 초기화한다.

```
this.manager = new SessionManager();
this.presence = new Presence();
```

제 8장, 데이터 다루기에서 데이터 소스를 처리하면서 데이터를 유지하기 위한 모델로 ListStore를 이용한 적이 있다. 이를 다시 구현해보자. 하지만 TreeView를 사용하는 대신 ComboBox를 이용할 수 있다. ComboBox 또한 MVC 디자인 패턴을 사용하며 ListStore를 모델로 이용한다. TreeView와 마찬가지로 화면에 사실상 데이터를 표시하기 위해선 렌더러가 필요하다. 다시 한 번 아래와 같은 코드 조각에 표시된 바와 같이 CellRendererText를 이용해 해당 작업을 실행한다.

```
var combo = ui.get_object("presenceStatus");
cell = new Gtk.CellRendererText();
combo.pack_start(cell);
combo.add_attribute(cell, "text", 1);
```

다음으로 ComboBox의 changed 시그널을 핸들러로 연결한다. 이 핸들러에서 우리가 하는 일은 ComboBox의 현재 Iter 객체를 얻고, iter 객체의 정수 값을 얻으며, Presence 프록시 객체의 SetStatusRemote를 호출하는 일이다. 앞서 논했듯이 Seed 코드에서 SetStatusRemote 함수는 사실상 D-Bus 측에서 SetStatus 함수다. 정수는 SetStatusRemote에서 전달하는데, 이는 인터페이스 객체에 대한 SetStatus 함수의 inSignature member에서 정의하였던 u의 값과 일치한다.

```
combo.signal.changed.connect(function(s) {
    var selected = {}
    s.get_active_iter(selected);
    var id = s.model.get_value(selected.iter, 0);
    self.presence.SetStatusRemote(id.value.get_int());
});
```

그 다음으로 textStatus 위젯의 변경된 시그널을 연결하고, 시그널을 수신하면 SetStatusTextRemote 함수를 호출한다. 이후 문자열에 해당하는 textStatus.text를 전달하는데, 이는 인터페이스 객체에서 정의한 SetStatus 함수의 inSignature에서 정의한 s 값과 일치한다.

```
var textStatus = ui.get_object("textStatus");
textStatus.signal.changed.connect(function(b) {
    self.presence.SetStatusTextRemote(textStatus.text);
});
```

리스트에서 다음으로 살펴볼 위젯은 logOut 위젯이다. LogoutRemoteSync를 호출함으로써 0 값을 넣어 clicked 시그널을 처리한다. GNOME Session Manager 문서에서 0은 단순히 로그아웃을 원한다는 의미다. 1은 프롬프트 없이 로그아웃 프로세스가 완료되었음을 의미한다. 마지막으로 값이 2인 경우 프롬프트 없이 모든 방해요소(inhibitor)를 무시하고 강제로 로그아웃이 이루어졌음을 뜻한다.

```
var logout = ui.get_object("logOut");
logout.signal.clicked.connect(function(b) {
    self.manager.LogoutRemoteSync(0);
});
```

다음으로는 powerOff 위젯을 살펴보겠다. 이 버튼을 누르면 종료(shutdown) 프로세스가 시작되길 원한다. 이를 위해선 ShutdownRemoteSync 함수를 호출한다.

```
var shutdown = ui.get_object("powerOff");
shutdown.signal.clicked.connect(function(b) {
    self.manager.ShutdownRemoteSync();
});
```

마지막으로 inhibit ToggleButton을 처리해야 한다. 버튼이 활성화되었을 때와 일반 상태일 때, 두 가지 상태를 처리한다. 활성화 상태에서는 라벨을 Unhibit으로 변경하여 행위가 변경되었음을 표시하길 원하고, SessionManager에게 우리를 방해요소(inhibitor)로 등록하길 요청한다. 방해요소가 있으면 어떠한 로그아웃 프로세스든 취소될 것이다. 이를 위해서는 InhibitRemoteSync를 호출한다. 리턴값은 우리가 inhibit으로 전달해야 하는 쿠키다. 이 쿠키는 ToggleButton과 반대의 상태로 UninhibitRemoteSync를 호출할 때 사용한다. 호출이 완료되면 로그아웃 프로세스는 다시 정상으로 돌아갈 것이다.

```
var inhibit = ui.get_object("inhibit");
inhibit.signal.toggled.connect(function(b) {
    if (inhibit.active == 1) {
        inhibit.label = "Uninhibit";
        var window = ui.get_object("window1");
        var xid = window.get_window().get_xid();
        self.inhibitCookie = self.manager.InhibitRemoteSync(application
 Id, xid, "I forbid you to logout", 1);
    } else {
        self.manager.UninhibitRemoteSync(self.inhibitCookie);
        inhibit.label = "Inhibit";
    }
```

```
});
```

이제 Vala 코드를 살펴보자. 코드가 어떻게 작동하는지는 Seed 코드와 동일하지만 D-Bus 함수를 어떻게 호출하는지를 살펴보자. 먼저 사용하길 원하는 메서드와 클래스의 인터페이스를 생성할 필요가 있다.

앞서 언급하였던 DBus 속성에 연결명이 따라온다.

```
[DBus (name = "org.gnome.SessionManager")]
```

인터페이스를 정의하고 GLib.Object를 파생하여 시그널도 처리할 수 있도록 한다. 각 메서드마다 camel case(각 단어의 첫 문자는 대문자로 쓰고 단어는 구분 문자 없이 모두 결합된다)로 쓴 원래 함수명을 밑줄이 결합된 소문자 단어로 변환한다. 가령 CanShutdown 함수는 can_shutdown으로 변환되어야 한다. 각 메서드는 IOError 예외를 던지도록 선언되어야 하며, 인자는 Vala 네이티브 데이터 타입을 이용해 Vala 스타일로 쓰여져야 한다.

```
interface SessionManager : GLib.Object {
    public abstract bool can_shutdown () throws IOError;
    public abstract void logout (uint32 mode) throws IOError;
    public abstract void shutdown () throws IOError;
    public abstract uint32 inhibit (string appId, uint32 xid, string
reason, uint32 flags) throws IOError;
    public abstract void uninhibit (uint32 cookie) throws IOError;
}
```

인터페이스를 사용하기 전에 인터페이스를 모두 선언하므로 Presence 인터페이스와 SessionManager는 아래와 같이 선언되어야 한다.

```
[DBus (name = "org.gnome.SessionManager.Presence")]
interface Presence: GLib.Object {
    public abstract void set_status_text (string text) throws IOError;
    public abstract void set_status (uint32 mode) throws IOError;
}
```

다음으로 이러한 인터페이스의 프록시 객체를 생성할 필요가 있다. get_proxy_sync를 사용해 이러한 인터페이스의 경로와 연결명을 매핑한다. null을 제외한 결과는 사용할 준비가 된 것이다.

```
manager = Bus.get_proxy_sync(BusType.SESSION,
    "org.gnome.SessionManager",
    "/org/gnome/SessionManager");

presence = Bus.get_proxy_sync(BusType.SESSION,
    "org.gnome.SessionManager",
    "/org/gnome/SessionManager/Presence");
```

지금쯤이면 manager 객체와 presence 객체는 D-Bus 객체로 연결될 준비가 되었을 것이다.

Seed에 비해 Vala에서 메서드를 호출하기는 매우 간단하다. 함수명을 직접 호출해 인자가 있다면 인자를 전달하면 될 일이다. 예를 들어 shutdown 함수는 아래의 코드 조각만으로 호출할 수 있겠다.

```
manager.shutdown();
```

시도해보기 - **null** 확인하기

앞서 언급했듯이 모든 프록시 객체가 D-Bus로 연결되도록 보장되는 것은 아니다. 예를 들어, 특정 연결이나 경로의 서비스 제공자가 설치되어 있지 않으면 연결은 실패하고 변수값은 null이 될 것이다.

우리 코드를 다시 살펴보고 프록시 객체의 발생마다 확인해보길 바란다. 값이 null이 아니면 코드는 그대로 사용해도 좋지만 null 이라면 어떤 조치든 취해야 한다. 이러한 경우를 처리할만한 전략을 세워보라.

런처

런처는 사용자가 아이콘을 클릭하여 애플리케이션을 실행시키는 장소다. GNOME Shell에서 런처는 **Activities** 메뉴의 **Applications** 탭에서 접근할 수 있다. 애플리케이션은 범주에 따라 열거되고, 애플리케이션의 아이콘, 제목, 설명을 표시한다. 제목과 설명은 잘 구분되어(localized) 있다. 애플리케이션은 검색을 통해 필터링할 수도 있다.

GNOME의 이전 버전이나 GNOME Fallback 모드에서는 런처가 GNOME 패널에 위치한다. 애플리케이션은 역시나 범주적으로 열거되며 설명 또한 구분되어 있다.

제 9장, GNOME을 통해 HTML5 애플리케이션 활용하기에서 애플리케이션을 열거하는 방식을 살펴봤지만 GNOME Shell이나 GNOME 패널과 정확히 같지는 않다. 실험에서는 데스크톱 파일에 포함된 정보를 로딩하는 작업이 꼭 필요했다. 따라서 애플리케이션의 아이콘을 런처에 표시되도록 만들기 위해서는 그것을 위한 데스크톱 파일을 생성할 필요가 있겠다.

실행하기 - 애플리케이션을 런처에 넣기

이제 앞의 애플리케이션에 런처를 생성해보자.

1. session.js를 session-tester로 재명명하라 (확장자 없이).
2. /usr/share/session-manager-test/session.ui를 열기 위해 UI 로더 부분을 수정하고, 아래의 행을 수정하라.

```
ui.add_from_file("session.ui");
```

3. 위의 행을 아래로 수정하라.

```
ui.add_from_file("/usr/share/session-manager-test/session.ui");
```

4. 새로운 텍스트 파일을 준비해 session-manager-test.desktop으로 명명하고 아래의 코드로 채워라.

```
[Desktop Entry]
Name=Session Manager Test
Comment=Testing the interaction with GNOME session manager
OnlyShowIn=GNOME;
Exec=session-tester
Icon=help-browser
StartupNotify=true
Terminal=false
Type=Application
Categories=GNOME;GTK;Settings
```

5. UI 파일을 /usr/share/session-manager-test 디렉터리에 설치하고 (디렉터리를 생성하지 않았다면 지금 생성하라!), 데스크톱 파일은 /usr/share/applications에, session-tester 스크립트는 /usr/bin에 설치하라.

6. GNOME Shell 에서 **Activities** 메뉴를 열고 **Applications** 탭에서 **Session Manager Test** 를 찾아라!

무슨 일이 일어났는가**?**

지금까지 시스템 전체 설정에서 UI 파일을 로딩하도록 스크립트를 간단히 수정해보았다. 이러한 과정은 애플리케이션이 배포된 곳마다 실행되어야 하는데, 이 예제는 UI 파일의 경로를 하드코딩하기 때문에 바람직한 예제는 아니다. 경로의 하드코딩을 피하는 한 가지 방법은 설정 파일을 보관하는 것이다. 제 4장, GNOME 코어 라이브러리 사용하기로 돌아가 설정 시스템으로 접근해보자.

그 다음 아래를 이용해 데스크톱 파일을 생성한다.

```
[Desktop Entry]
```

이후 애플리케이션명을 명시한다. 그 다음은 런처에 표시될 텍스트를 명시한다.

```
Name=Session Manager Test
Comment=Testing the interaction with GNOME session manager
```

OnlyShowIn은 해당 애플리케이션이 표시될 런처를 정의한다. 가령 Unity를 추가하면 애플리케이션은 Unity와 GNOME에서만 표시될 것이다.

```
OnlyShowIn=GNOME;
```

다음으로 실행 파일명을 정의한다. /usr/bin으로 설치하는 애플리케이션명은 다음과 같다.

```
Exec=session-tester
```

이후 런처에서 사용되는 아이콘을 정의한다. 지금은 help-browser 애플리케이션에서 아이콘을 훔치겠다. 실제 애플리케이션에서는 우리만의 아이콘을 제공해야 한다.

```
Icon=help-browser
StartupNotify=true
```

그리고 나면 애플리케이션을 실행하는 데에 terminal이 필요하지 않음을 명시한다. 가령, bash 스크립트를 제공할 경우, hashbang이 없거나 실행 파일의 권한이 누락되면 terminal에게 해당 스크립트를 열도록 요청해야 할 수도 있다.

```
Terminal=false
```

다음으로 애플리케이션을 Application으로 정의한다.

```
Type=Application
```

마지막으로 애플리케이션을 이러한 범주로 나누어야 함을 런처에게 알린다. 애플리케이션이 Settings로 매핑하는 **System Tools** 에 상주함을 런처에서 간단히 확인할 수 있다.

```
Categories=GNOME;GTK;Settings
```

런처는 보통 /usr/share/applications 내의 모든 파일에 적용시킨 변경내용을 전부 듣는다. 따라서 새로운 파일을 넣거나 기존 파일을 수정할 때마다 변경내용은 즉시 런처로 자동 표시된다고 할 수 있기 때문에 데스크톱을 재시작할 필요가 없다.

데스크톱 파일의 내용에 대한 형식 명세서는 http://standards.freedesktop.org/desktop-entry-spec/latest/index.html 에서 찾을 수 있다.

GNOME Keyring

실제 애플리케이션을 사용할 때는 비밀번호, 비밀 데이터 또는 키를 저장하는 경우가 많다. 이러한 유형의 데이터 저장공간을 구현하기란 까다로우며, 보안 분야에 특별한 기술을 요한다. GNOME Keyring은 GNOME 플랫폼에서 이용할 수 있는 비밀 데이터 저장공간 구조에 해당한다. GNOME Keyring을 사용하는 애플리케이션은 비밀번호, 비밀 데이터 또는 키를 키링으로 저장하여 추후 필요할 때 검색할 수 있다.

키링은 보호되어 있으며 사용자 계정과 연계된다. 사용자가 시스템으로 로긴할 때마다 자동으로 애플리케이션에서 키링을 이용하고 사용자가 로그아웃하면 키링 또한 닫히도록 설정할 수도 있다. 아니면 비밀번호를 이용해 키링을 열 수도 있다.

키링에는 **Seahorse** 라고 불리는 애플리케이션이 따라온다. 이 애플리케이션은 저장된 비밀 데이터를 사용자의 세션 내에서 표시한다. 우리가 목표로 한 통합은 Seahorse를 대체하는 것이 아니라 키링에 데이터를 안전하게 보관하는 것이다. 하지만 Seahorse 를 이용해 데이터를 검사할 수는 있겠다.

실행하기 - 비밀번호를 안전하게 저장하기

불행히도 비밀번호를 안전하게 저장하기 위해서는 Vala를 이용해야만 하는데, Seed가 필요로 하는 gir-1.2에는 비밀번호를 쉽게 저장하는 데에 필요한 함수가 포함되어 있지 않기 때문이다.

1. GtkBuilder 없이 빈 Vala 객체를 새로 생성하고 keyring이라 부른다.

2. configure.ac를 열고 PKG_CHECK_MODULES(KEYRING, [gtk-3.0])를 찾아라. 전체 행을 아래의 행으로 수정하라.

```
PKG_CHECK_MODULES(KEYRING, [gnome-keyring-1 ])
```

3. src/Makefile.am을 열고 아래의 행을 찾아라.

```
keyring_SOURCES = \
    keyring.vala config.vapi

keyring_VALAFLAGS = \
    --pkg gtk+-3.0
```

4. 위의 내용을 모두 아래의 내용으로 수정하라.

```
keyring_SOURCES = \
    keyring.vala config.vapi gnome-keyring.vapi

keyring_VALAFLAGS = \
    --pkg gnome-keyring-1
```

5. src/ 디렉터리에 새로운 파일을 생성하고 gnome-keyring.vapi라고 명명한 다음 아래의 내용으로 채워라.

```
[CCode (cprefix = "GnomeKeyring", lower_case_cprefix =
"gnome_keyring_")]
namespace GnomeKeyringOverrides {
[Compact]
public struct PasswordSchemaAttribute {
    public unowned string name;
        public GnomeKeyring.AttributeType type;
            }

            [Compact]
            [CCode (cheader_filename = "gnome-keyring.h")]
            public struct PasswordSchema {
                public GnomeKeyring.ItemType item_type;
                [CCode(array_length = false)]
                public PasswordSchemaAttribute[] attributes;
            }
            [CCode (cheader_filename = "gnome-keyring.h")]
            public static void* store_password (GnomeKeyringOverrides.
PasswordSchema schema,
string? keyring, string display_name, string password, owned
GnomeKeyring.OperationDoneCallback callback, ...);

            [CCode (cheader_filename = "gnome-keyring.h")]
            public static void* find_password (GnomeKeyringOverrides.
PasswordSchema schema,
```

```
owned GnomeKeyring.OperationGetStringCallback callback, ...);
}
```

6. src/keyring.vala를 열고 아래의 코드를 이용하라.

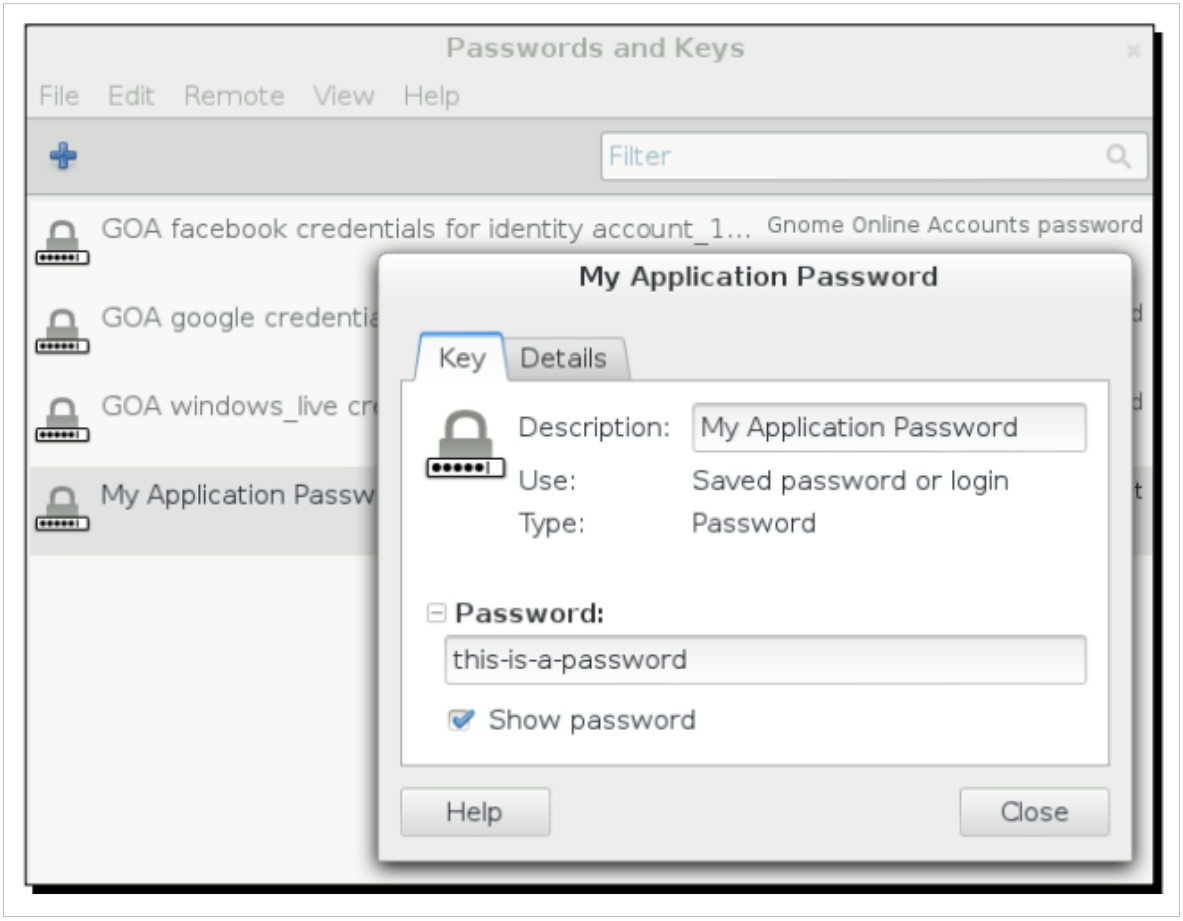
```
using GnomeKeyring;

public class Main : Object
{
    private const GnomeKeyringOverrides.PasswordSchema secretData =
{
        ItemType.GENERIC_SECRET,
        {
            { "name", AttributeType.STRING },
            { null, 0}
        }
    };

    public void returning_password_callback(Result result, string?
password) {
        if (result == Result.OK) {
            stdout.printf ("Password is: %s\n", password);
        } else {
            stdout.printf ("Failed, code: %d\n",(int) result);
        }
    }

    public void store_password_callback(Result result) {
        if (result == Result.OK) {
            GnomeKeyringOverrides.find_password ( secretData,
returning_password_callback, "name", "myuser", null);
        } else {
            stdout.printf ("Failed, code: %d\n",(int) result);
        }
    }

    public Main ()
    {
        GnomeKeyringOverrides.store_password ( secretData, null, "My
Application Password", "this-is-a-password",
store_password_callback, "name", "myuser", null);
    }

    static int main (string[] args)
    {

        var app = new Main ();
```

```
        var loop = new MainLoop();
            loop.run ();
        return 0;
    }
}
```

7. 애플리케이션을 빌드하되 아직 실행하지 말라!
8. GNOME Shell에서 **Activities** 메뉴를 열고 **Seahorse** 애플리케이션을 찾아 실행하라. 안에 몇 개의 엔트리가 보일 것인데 (어쩌면 비어있을 수도 있다), **My Application Password** 란 이름의 엔트리는 보이지 않을 것이다. Seahorse를 종료하라.
9. 이제 애플리케이션을 실행하면 아래를 표시할 것이다.

```
Password is: this-is-a-password
```

10. 계속해서 실행되지만 조금 지나면 Ctrl+C 키조합을 이용해 종료할 수 있다.
11. Seahorse를 다시 실행하면 **My Application Password** 라는 엔트리가 보일 것이다. 엔트리를 클릭하면 비밀번호, 즉 **this-is-a-password** 를 볼 수 있다.



무슨 일이 일어났는가**?**

사용자 비밀번호를 GNOME Keyring 시스템으로 저장한 것이다. 이 비밀번호는 앞절에서 애플리케이션을 이용해 보인 바와 같이 후에 필요할 때 복구할 수 있다. 웹 브라우저에서 "비밀번호 기억(remember password)" 기능을 사용하는가? 이 기능을 사용하면 브라우저는 비밀번호 필드를 저장된 비밀번호로 미리 채워 놓는다. 하지만 이번 예제에서는 비밀번호를 자신만 알고 숨겨두는 대신 GNOME Keyring에 저장하는 셈이 된다.

안타깝게도 많이 사용되는 배포판에 배포되는 GNOME Keyring에 대한 Vala 함수 및 클래스 매핑은 올바르지 않은 정보를 갖고 있기 때문에 그대로 사용할 수 없는 노릇이다. 제 9장, GNOME을 통해 HTML5 활용하기에서는 원본 .vapi 파일을 커스텀 .vapi 파일로 대체하는 방법을 보인 적이 있다. 하지만 지금은 전체 파일을 대체하는 대신 올바르지 않은 정보를 수정하는 오버라이드 파일을 넣는다. gnome-keyring.vapi 라고 부르는 오버라이드 파일은 앞서 실행하기 - 비밀번호 안전하게 저장하기 절의 4번 단계에서 살펴본 것처럼 빌드 단계에 포함된다.

오버라이드 파일의 내용은 올바르지 않게 작성된 원본 버전을 대체하는 함수 및 클래스 member에 해당한다. 이번 예제에서는 store_password와 find_password 함수, 그리고 PasswordSchema 구조체에 대한 오버라이드를 갖고 있다. 원본 버전을 무작정 대체하기보다는 다른 네임스페이스를 이용할 필요가 있다. 이번 예제에서는 GnomeKeyringOverrides 네임스페이스를 사용한다. 따라서 원본 버전을 사용하길 원할 때마다 GnomeKeyring을 사용하고, 오버라이드된 버전을 사용하고 싶으면 GnomeKeyringOverrides 네임스페이스를 사용한다.

store_password와 find_password 함수를 사용하기 전에는 보관하길 원하는 데이터의 구조체를 정의해야 한다. 아래 코드 조각에서 우리는 secretData라고 부르는 구조체를 설명한다. 데이터에는 name이라는 하나의 필드만 있다. 이 데이터를 이용해 우리는 이름과 연관된 비밀번호를 저장하길 원한다. null과 0의 쌍을 구조체의 끝으로 표시하는 구조체를 종료한다.

```
private const GnomeKeyringOverrides.PasswordSchema secretData = {
    ItemType.GENERIC_SECRET,
    {
        { "name", AttributeType.STRING },
        { null, 0}
    }
};
```

앞의 예제에서는 비동기식 버전의 저장소를 사용해 비밀번호 함수를 찾는다. 즉, 이 함수를 호출하면 즉시 함수로부터 리턴한다는 뜻이다. 함수의 성공은 다른 콜백 함수에서 처리한다. 동기식 버전을 (함수명이 sync로 끝난다) 선택하였는데 GNOME Keyring 측에서 부담스러운 무언가가 실행 중이라면 연산이 완료될 때까지 애플리케이션이 freeze할 것인데, 이러한 일은 비동기식 호출에선 발생하지 않을 것이다.

앞에서 언급했듯 몇 가지 콜백 함수를 준비해야 한다. 첫 번째는 find_password 함수를 위한 것이다. result 변수에 결과를 얻고 password 변수에서 비밀번호를 얻는다. 비밀번호 타입 정의에서 물음표는 내용이 null일 수 있다는 의미임을 명심하라. 결과가 Result.OK 이외의 값일 때마다 이런 일이 발생한다. 함수는 실패 시 비밀번호 또는 오류 코드를 출력하기 위해 사용된다. 실제 애플리케이션에서는 리턴된 비밀번호를 필더 또는 비밀번호 내용을 필요로 하는 다른 요소로 넣을 것이다.

```
public void returning_password_callback(Result result, string?
password) {
    if (result == Result.OK) {
        stdout.printf ("Password is: %s\n", password);
    } else {
        stdout.printf ("Failed, code: %d\n",(int) result);
    }
}
```

다음으로 store_password 함수를 위한 콜백 함수가 있다. 이는 result 값을 확인하는데, 성공적인 값이면 find_password 함수로 비밀번호를 얻기를 시도하면 된다. 실제 애플리케이션에서는 비밀번호가 저장되었다는 알림을 표시하거나 아니면 조용히 다른 실행을 할 것이다.

```
public void store_password_callback(Result result) {
    if (result == Result.OK) {
        GnomeKeyringOverrides.find_password ( secretData,
returning_password_callback, "name", "myuser", null);
    } else {
        stdout.printf ("Failed, code: %d\n",(int) result);
    }
}
```

메인 함수에서는 **My Application Password** 라고 불리는 영역에 **this-is-a-password** 비밀번호를 저장한다. 비밀번호를 myuser 문자열과 연관시킨다. 실제 애플리케이션에서 이러한 연관은 확장이 가능하다. 가령 네트워크 비밀번호의 경우 필요한 추가 데이터를 서버명, 포트 번호, 서비스 경로 등으로 확장이 가능하다. 데이터의 끝을 표시하기 위해 함수는 null로 끝난다. 확장된 데이터 구조체가 있다면 구조체에 정의된 순서대로 필요한 데이터를 계속해서 함수의 매개변수로 전달하고 null로 끝내야 한다.

```
public Main ()
{
    GnomeKeyringOverrides.store_password ( secretData, null, "My
Application Password", "this-is-a-password",
store_password_callback, "name", "myuser", null);
}
```

알림 시스템

애플리케이션은 지금 발생 중인 이벤트에 대해 사용자에게 통지해야 하는 경우가 있다. 애플리케이션이 활성화되었고 사용자가 현재 사용 중이라면 간단히 정보를 애플리케이션 안에 표시할 수 있다. 하지만 애플리케이션이 현재 실행 중이지만 바탕화면에서(in the background) 유지되거나 크기가 최소화되었다면 어떤 일이 발생할까? 사용자가 그 정보를 보지 못하는 일이 발생할 것이다. 이는 바람직하지 못하며, 특히 매우 중요한 정보를 표시한다면 더 그러하다. 이 때를 대비해 알림 시스템을 사용해야 하는데, 사용자는 언제든지 사용할 수 있다.

GNOME 에는 **libnotify** 라는 알림 시스템이 있다. 메인 프로세스는 절대 종료되지 않고 데스크톱이 종료되어야만 종료되는 **daemon** 이라는 프로그램으로서 실행된다. 이 프로그램은 알림을 표시하라는 모든 애플리케이션의 요청을 듣는다. 요청을 수신하면 화면 하단에 알림 텍스트를 표시한다. Application은 단순히 라이브러리를 이용해 알림을 데몬으로 전송한다.

실행하기 - 알림 전송하기

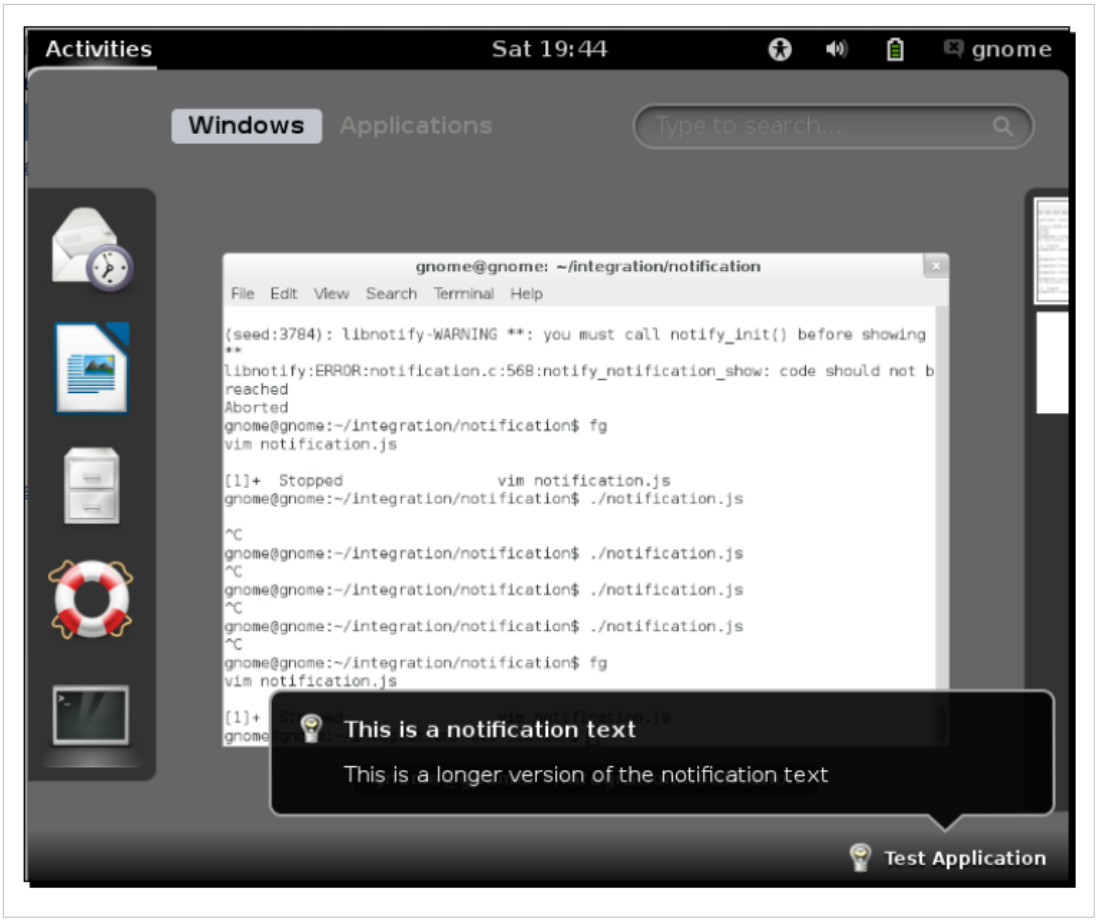
몇 가지 알림을 전송해보자.

1. notification.js라는 새로운 Seed 스크립트를 생성하라.
2. 스크립트를 아래의 코드로 채워라.

```
#!/usr/bin/env seed

GLib = imports.gi.GLib;
```

```
Notify = imports.gi.Notify;
GObject = imports.gi.GObject;

Main = new GType({
    parent: GObject.Object.type,
    name: "Main",
    init: function() {
        Notify.init('Test Application');
        var n = new Notify.Notification({
            summary: 'This is a notification text',
            body: 'This is a longer version of the notification text',
            });
        n.add_action('ok-button', 'OK', function() { n.close()});
        n.show();
    }
});

var main = new Main();
var context = GLib.main_context_default();
var loop = new GLib.MainLoop.c_new(context);
loop.run();
```

3. 이를 실행하라. 알림 텍스트의 끝이 잘려 표시되지 않음을 눈치챌 것이다. 화면 하단에 잠깐 동안 표시되는데, 텍스트로 마우스를 갖다 대면 텍스트 상자가 확장되어 모든 텍스트가 표시될 것이다. **OK** 버튼도 표시할 것이다. 하지만 해당 영역에서 벗어나자마자 텍스트는 사라질 것이다. **OK** 버튼을 누르지 않으면 알림은 화면의 하단 우측 모서리에 남아있다.



무슨 일이 일어났는가**?**

우리는 단순히 libnotify로 호출하는 일만 실행했을 뿐이다. 먼저 Notify로부터 가져오기를 실행함으로써 선언할 필요가 있다.

```
Notify = imports.gi.Notify;
```

libnotify로부터 함수를 호출하기 전에 먼저 init 함수를 호출해야 한다. 그렇지 않으면 어떤 함수도 성공적으로 호출할 수 없다.

```
Notify.init('Test Application');
```

다음으로 새로운 알림 객체를 생성한다. 각 객체마다 하나의 알림 텍스트를 보유할 수 있다. 생성자에 텍스트, 개요(summary), 본문(body)을 설정한다. summary 부분은 짧은 텍스트로, 화면에 처음 표시될 것이다. body 부분은 개요보다 조금 긴 텍스트인데, 알림 영역 안에 마우스 커서를 위치시키면 표시된다. Seed에서는 이 텍스트를 객체로 전달할 수 있다.

```
var n = new Notify.Notification({
    summary: 'This is a notification text',
    body: 'This is a longer version of the notification text',
});
```

다음으로 OK 버튼을 추가한다. 버튼이 클릭될 때마다 알림을 닫으면 된다. 액션은 아래 정의된 것처럼 익명(anonymous) 함수를 이용해 추가된다.

```
n.add_action('ok-button', 'OK', function() { n.close()});
```

알림을 표시하고 싶으면 show 함수를 호출하기만 하면 된다.

```
n.show();
```

알림은 프로그램이 닫혀도 알림 영역에 계속 남는데, **OK** 버튼을 눌러야만 닫을 수 있다.

시도해보기 - 아이콘 표시하기

알림 안에 아이콘을 표시해보자. 이는 구현하기가 매우 쉬운데, 생성자에서 전달하는 객체의 icon 필드에 아이콘명을 넣으면 끝이다.

요약

이번 장에서는 데스크톱의 몇 가지 중요한 부분, 즉 세션 관리, 런처, 키링, 알림 시스템으로의 통합에 대해 학습해보았다. D-Bus에 대해서도 소개했다. 또 일반적인 라이브러리 API 뿐만 아니라 D-Bus를 이용하는 기능으로 접근하는 방법도 논했다.

이제 로그아웃 또는 종료 프로세스를 시작할 수 있는 애플리케이션을 생성하는 방법과, 세션 관리자가 프로세스를 계속하지 못하도록 만드는 방법을 이해한다. /usr/share/applications 디렉터리에 데스크톱 파일을 설치함으로써 런처에 애플리케이션을 표시하는 방법도 학습했다. 민감한 데이터를 키링에 저장하고 필요 시 찾아오는 방법도 배웠다. 마지막으로, 데스크톱에 표시되는 알림의 표시 방법에 대해서도 논했다. 이러한 경험은 애플리케이션을 GNOME 데스크톱과 훌륭하게 통합되도록 만드는 데에 도움이 될 것이다.

또 많이 사용되는 배포판에서 제공되는 올바르지 않은 .vapi 파일 문제를 오버라이드 파일을 생성함으로써 해결하는 방법도 배웠다.

다음 장에서는 애플리케이션이 전세계에서 사용될 경우 집중해야 할 측면들을 논하고자 한다. 따라서 본인의 애플리케이션이 국제적 수준에서 성공하길 원한다면 놓쳐선 안 될 내용이다.

Notes

GNOME3ApplicationDevelopmentBeginnersGuide:Chap 12

제 12 장 품질이 좋으면 모든 게 쉬워진다

품질이 좋으면 모든 게 쉬워진다

소프트웨어 품질은 제품을 시장으로 발표하기 전에 확인하는 것이 아니다. 품질의 확인은 그보다 훨씬 전에 이루어진다. 많은 소프트웨어 개발자들은 한 조각의 코드가 쓰여지기도 전에 품질을 확인하기도 한다. 테스트를 거치고 잘 작성됨과 동시 잘 관리된 코드와 잘 정의된 규칙 집합은 소프트웨어 개발에 필수적이다. 이 모두를 GNOME 개발 환경에서 어떻게 실현할 수 있을까? 한 번 알아보자.

소프트웨어에 필요한 여러 테스팅 중에서 단위 테스팅은 종종 쓰기가 까다롭다. 따라서 이번 장에서는 단위 테스팅의 실행을 집중적으로 살펴볼 것이다. GLib, GTK+, Gdk가 제공하는 테스팅 프레임워크를 사용할 예정이다. 본 장에서는 테스팅을 쉽게 자동화하는 방법에 관한 통찰력을 얻을 수 있도록 Anjuta 대신 명령행을 직접적으로 사용할 것이다. 이번 장에 실린 활동에서는 앞 장의 여러 장에 걸쳐 사용된 코드를 사용하고 그에 단위 테스팅을 추가하고자 한다.

구체적으로 다룰 내용은 다음과 같다.

- 단위 테스팅의 개념
- 라이브러리 스터빙(stubbing)
- GUI 모듈 테스트하기

그럼 본격적으로 시작해보자.

단위 테스팅을 실행하는 이유

단위 테스트는 소스 코드에서 특정 객체를 대상으로 실행되는 구체적인 테스트로서 국소화되어 실행된다. 테스트가 각 기능을 실제로 다루는지 확실히 할 필요가 있다. 단위 테스팅은 아래와 같이 간단하게 이루어진다.

- 각 함수를 테스트한다.
- 함수 내 각 branch를 테스트한다. 함수에 if와 else문, 또는 switch와 case문이 포함된 경우를 예로 들 수 있겠다.
- 앞의 규칙들은 소스 코드에서 모든 장소를 확보하여 프로그램을 소비자에게 전달할 때 놀랄만한 요소, 특히 바람직하지 못한 요소가 기다리고 있지 않도록 확보하기 위함이다.
- 함수를 테스트하기 위해서는 인자 리스트에 데이터를 전달할 필요가 있다. 우선 데이터를 만들고, 필요하다면 두 번째 규칙이 충족되도록 여러 데이터 집합을 이용해 테스트를 반복한다.
- 단위 테스팅은 코드 조각을 쓸 때마다 실행된다. 많은 소프트웨어 개발자들은 코드를 쓰기도 전에 단위 테스트를 생성함으로써 "Test-Driven Development(TDD)"(테스트 주도 개발) 방법론을 사용한다! 이런 경우 코드로부터 예상되는 내용을 이미 알고 있기 때문에 어떤 함수를 전달해야 하는지에 초점을 둘 수 있다는 장점이 있다. 이는 특히 라이브러리를 작성할 때 중요한데, 단위 테스트를 먼저 생성하여 API가 어떻게 생겼는지와 어떻게 사용할 것인지를 정확히 알 수 있기 때문이다.
- 단위 테스팅은 결정적(deterministic)이어야 하는데, 이는 특정 입력을 제공함으로써 출력이 무엇인지 정확히 알아야 한다는 의미다. 이를 성공적으로 구현하기 위해서는 테스팅에 데이터를 무작위로 집어넣어선 안 되며, 우리가 사용하는 API의 행위 역시 결정적이어야 한다.

실행하기 - 첫 번째 단위 테스트 생성하기

첫 활동으로, 맨 처음으로 Vala를 다루었던 코드인 제 3장, 프로그래밍 언어의 Vala 알아가기 절에서 소개한 hello-vala 프로그램을 살펴보자. 단위 테스트를 생성하게 될 두 개의 객체, Book과 Bookstore가 있다.

1. Anjuta에서 hello_vala.anjuta를 다시 열어보자.

2. 프로젝트의 최상위 디렉터리에서 configure.ac 파일을 열어라. 아래 코드에 표시된 출력 부분을 찾아라.

```
AC_OUTPUT([
Makefile
src/Makefile
tests/Makefile
])
```

3. 아래 코드와 같이 변경하라.

```
AC_OUTPUT([
Makefile
src/Makefile
tests/Makefile
])
```

4. 프로젝트 최상위 디렉터리에서 Makefile.am 파일을 열고 SUBDIRS 부분을 찾아라.

```
SUBDIRS = src
```

5. 아래 코드와 같이 변경하라.

```
SUBDIRS = src tests
```

6. 최상위 디렉터리에 tests라는 디렉터리를 생성하라.
7. tests 디렉터리에 새로운 Makefile.am 파일을 생성하고 아래의 코드로 채워라.

```
AM_CPPFLAGS = \
    -DPACKAGE_LOCALE_DIR=\""$(localedir)"\" \
    -DPACKAGE_SRC_DIR=\""$(srcdir)"\" \
    -DPACKAGE_DATA_DIR=\""$(pkgdatadir)"\" \
    $(HELLO_VALA_CFLAGS) \
    $(atk_CFLAGS) \
    $(gee-1.0_CFLAGS)
AM_CFLAGS =\
    -Wall\
    -g
TESTS=test_book test_bookstore
check_PROGRAMS = test_book test_bookstore
test_book_SOURCES = \
    test_book.vala ../src/book.vala
test_book_VALAFLAGS = \
    --pkg gtk+-3.0 \
    --pkg gee-1.0
test_book_LDFLAGS = \
    -Wl,--export-dynamic
test_book_LDADD = $(HELLO_VALA_LIBS) \
    $(atk_LIBS) \
```

```
    $(gee-1.0_LIBS)
test_bookstore_SOURCES = \
    test_bookstore.vala ../src/book.vala ../src/bookstore.vala
test_bookstore_VALAFLAGS = \
    --pkg gtk+-3.0 \
    --pkg gee-1.0
test_bookstore_LDFLAGS = \
    -Wl,--export-dynamic
test_bookstore_LDADD = $(HELLO_VALA_LIBS) \
    $(atk_LIBS) \
    $(gee-1.0_LIBS)
```

8. tests 디렉터리에 test_book.vala라는 새로운 Vala 코드를 생성하고 아래의 코드로 채워라.

```
public class TestBook {
    static void test_isbn ()
    {
        var b = new Book("1", "title");
        assert(b.isbn == "1");
    }
    static void test_title ()
    {
        var b = new Book("1", "title");
        assert(b.title == "title");
    }
    static void test_add_author()
    {
        var b = new Book("1", "title");
        b.addAuthor("author1");
        b.addAuthor("author2");
        b.addAuthor("author3");
        assert(b.authors.size == 3);
    }
    static int main (string[] args)
    {
        Test.init (ref args);
        Test.add_func ("/test-isbn", test_isbn);
        Test.add_func ("/test-title", test_title);
        Test.add_func ("/test-add-author", test_add_author);
        Test.run ();
        return 0;
    }
}
```

9. test_bookstore.vala라는 새로운 Vala 파일을 생성하고 tests 디렉터리에 넣어라. 아래 코드를 파일로 복사하라.

```
public class TestBookStore {
    static void test_add_stock()
    {
        var b = new Book("1", "title");
        var s = new BookStore(b, 1.0, 12);
        s.addStock(13);
        assert(s.getStock() == 25);
        assert(s.isAvailable() == true);
    }
    static void test_remove_stock()
    {
        var b = new Book("1", "title");
        var s = new BookStore(b, 1.0, 12);
        s.addStock(13);
        s.removeStock(10);
        assert(s.getStock() == 15);
        assert(s.isAvailable() == true);
    }
    static void test_stock_alert()
    {
        var b = new Book("1", "title");
        var s = new BookStore(b, 1.0, 12);
        var alert_emitted = false;
        s.stockAlert.connect(() => {
            alert_emitted = true;
        });
        s.removeStock(1);
        assert(alert_emitted == false);
        s.removeStock(10);
        assert(alert_emitted == true);
    }
    static void test_price_alert()
    {
        var b = new Book("1", "title");
        var s = new BookStore(b, 1.0, 12);
        var alert_emitted = false;
        s.priceAlert.connect(() => {
            alert_emitted = true;
        });
        s.setPrice(2.5);
        assert(alert_emitted == false);
        s.setPrice(0.5);
        assert(alert_emitted == true);
    }
    static int main (string[] args)
    {
        Test.init (ref args);
```

```
        Test.add_func ("/test-add-stock", test_add_stock);
        Test.add_func ("/test-remove-stock", test_remove_stock);
        Test.add_func ("/test-stock-alert", test_stock_alert);
        Test.run ();
        return 0;
    }
}
```

10. src/book.vala를 열고 파일에서 아래의 코드 조각을 찾아라.

```
private string title;
private string isbn;
private ArrayList<string> authors;
```

11. 위의 코드를 아래로 대체하라.

```
internal string title;
internal string isbn;
internal ArrayList<string> authors;
```

12. 프로젝트의 최상위 디렉터리에서 terminal를 통해 아래의 명령을 발행하라.

```
./autogen.sh
```

13. 프로젝트를 빌드하고 아래의 명령을 입력하여 모두 문제없이 작동하는지 확인하라.

```
make all
```

14. 아래 명령을 실행하여 단위 테스트를 만들어 실행하라.

```
make check
```

15. 아래와 비슷한 내용이 출력되는지 확인하라.

```
/test-isbn: OK
/test-title: OK
/test-add-author: OK
PASS: test_book
/test-add-stock: OK
/test-remove-stock: OK
/test-stock-alert: OK
PASS: test_bookstore
==================
All 2 tests passed
==================
```

무슨 일이 일어났는가**?**

와우! 순식간에 완성했다.

테스트 경로와 테스트 결과를 표시함으로써 코드의 유효성을 검사하는 단위 테스트가 생겼다. 앞 절을 통해 테스트 경로 모두에게서 OK를 수신하였음을 확인하였을 것이다. 테스트에 대한 통계도 포함되어 있었다.

먼저 configure.ac와 Makefile.am을 수정함으로써 autotools 기본구조를 준비한 후에 tests 디렉터리에 새로운 Makefile.am 파일을 생성하였다. configure.ac의 output 섹션과 Makefile.am의 SUBDIRS 섹션 모두에 tests 디렉터리를 넣었다. 이들이 없다면 tests 디렉터리는 autotools와 나머지 빌드 기본구조에 알려지지 않을 것이다.

그러면 테스트 파일이 생긴다. Makefile.am에서 아래와 같은 행을 찾을 수 있을 것이다.

```
TESTS=test_book test_bookstore

check_PROGRAMS = test_book test_bookstore
```

이는 두 개의 테스트 프로그램, 즉 test_book과 test_bookstore가 있음을 autotools로 알려준다. 첫 행은 우리가 make check 명령을 발행하면 autotools가 명시된 프로그램을 실행하도록 만든다. 두 번째 행은 autotools에게 명시된 이름으로 된 바이너리를 생성하라고 말한다.

그러면 test_book과 test_bookstore에 해당하는 섹션이 생긴다. test_book에 관한 섹션에서는 다음과 같은 흥미로운 점이 발견된다.

```
test_book_SOURCES = \
    test_book.vala ../src/book.vala
```

이는 test_book 프로그램에 대한 소스 코드 파일이 test_book.vala와 book.vala임을 autotools로 알리는데, 이 파일들은 ../src 디렉터리에 위치한다. test_bookstore 프로그램에 대해서도 마찬가지다. 테스트 중인 소스 코드와 단위 테스트를 함께 컴파일하는 것이 보통이다.

이제 단위 테스트를 어떻게 쓰는지 확인해보자. 먼저 test_book.vala를 살펴보자.

```
public class TestBook {
```

각 단위 테스트마다 특정 객체를 테스트하는 클래스가 있다. 클래스는 객체명 앞에 Test라는 단어를 붙여 명명한다. 여기서 객체는 Book이므로 단위 테스트명은 TestBook이 되겠다. 이는 일반 클래스에 해당한다.

다음으로 테스트하고자 하는 객체에 실행하길 원하는 테스트를 static 함수에서 정의한다. 함수들이 static한 이유는 단위 테스트에서 test 클래스의 객체를 인스턴스화하지 않을 것이기 때문이다.

하나의 테스트만으로 관련된 사례를 모두 테스트해야 한다. 첫 번째 테스트에서는 ISBN이 올바로 설정되었는지 테스트하길 원한다. 여기서 ISBN 값이 1인 Book 객체를 생성하고, assert 함수를 이용해 isbn member의 값이 실제로 1인지 검사한다.

```
static void test_isbn ()
{
    var b = new Book("1", "title");
    assert(b.isbn == "1");
}
```

조금 어리석다고 생각할지도 모르겠다. 그렇게 빤한 것을 왜 굳이 검사하려 할까? 인간의 눈으로 볼 때는 항상 검사를 통과할 것이라고 생각할지도 모른다. 하지만 코드가 커질수록, 그리고 의도적으로 또는 뜻하지 않게 코드를 약간만 변경하는 경우 테스트가 실패할 수 있다는 사실을 명심하길 바란다. 필자의 주장을 강조하

기 위해 book.vala의 실제 코드를 살펴보도록 하겠다.

```
public Book(string isbn, string title) {
    this.isbn = isbn;
    this.title = title;
    authors = new ArrayList<string>();
}
```

테스트에서 우리 코드는 객체를 인스턴스화한 직후 isbn member를 확인하였다. 아래와 같이 생성자를 수정했다고 가정해보자.

```
public Book(string isbn, string title) {
    isbn = isbn;
    title = title;
    authors = new ArrayList<string>();
}
```

앞의 행에서 어쩌다가 this를 제거하는 일이 발생할지도 모른다. 코드는 여전히 실행되겠지만 결과를 틀리다. 테스트가 없이는 이를 간과하기 쉬우며, 결국 불평하는 고객에게 프로그램이 계산을 잘못 수행한다는 소식을 전해주어야 할지도 모른다! 이제 실제로 앞의 오류를 만들어 make check를 다시 실행해보자. 아래의 화면 출력에서 알 수 있듯이 즉시 통지를 받을 것이다.

```
/test-isbn: **
ERROR:test_book.c:100:test_book_test_isbn: assertion failed: (g_strcmp0
(_tmp1_, "1") == 0)
/bin/bash: line 5: 9305 Aborted ${dir}$tst
FAIL: test_book
/test-add-stock: OK
/test-remove-stock: OK
/test-stock-alert: OK
PASS: test_bookstore
====================
1 of 2 tests failed
====================
```

예상한대로 test_isbn 함수가 오류를 정확히 가리키고 있음을 확인할 수 있다. 이제 test_isbn이 없다고 가정해보자.

함수로 전달한 표현식의 올바른 값을 확인하기 위해 assert를 사용한다. assert 함수는 값이 true가 아닌 경우 즉시 프로그램을 종료할 것이기 때문에, 프로그램이 종료되면 우리는 무언가 잘못되었음을 알게 된다.

Book 클래스를 아래와 같은 모양이 되도록 수정할 필요가 있었다.

```
internal string title;
internal string isbn;
internal ArrayList<string> authors;
```

그리고 모든 private member를 internal로 변경했다. Vala는 같은 패키지 내 모든 클래스의 내부 member를 연다. 즉, 단위 테스트가 member로 직접 접근할 수 있다는 뜻이다. 그렇지 않으면 단위 테스트는 authors member 등으로 접근할 수 없을 것이다. C++에서는 friend 클래스라는 것이 있어서 이를 통해 member로 접근하는 동시 member를 private하게 유지할 수도 있는데, Vala는 이에 해당하는 기능이 없다는 점에서 짜증스럽기도 하다.

그 다음 행에서는 제목 값을 검사하는 테스트가 있다.

```
static void test_title ()
{
    var b = new Book("1", "title");
    assert(b.title == "title");
}
```

이는 test_isbn 테스트와 같은 방식으로 작동한다. 다음으로, addAuthor 함수가 그 기능에 맞게 작동하는지 검사하기 위한 테스트가 있다.

```
static void test_add_author()
{
    var b = new Book("1", "title");
    b.addAuthor("author1");
    b.addAuthor("author2");
    b.addAuthor("author3");
    assert(b.authors.size == 3);
}
```

여기서 authors 배열 리스트를 확인한다. 이는 addAuthor 함수를 이용해 세 명의 저자를 Book 객체로 추가한 후 3의 값을 가져야 한다. Book 객체에는 더 이상 테스트가 필요한 함수가 없다. 나머지 함수는 데이터를 화면으로 출력할 뿐이다. 그러므로 단위 테스트의 main 함수로 넘어가 보자.

```
static int main (string[] args)
{
    Test.init (ref args);
```

먼저 Test.init 함수를 호출함을 확인할 수 있다. 이것은 GLib.Test 클래스의 초기화 함수다.

다음으로 앞서 정의한 함수를 모두 등록하고, 함수에 테스트 경로를 할당한다. 테스트 경로는 실제 테스트 함수를 보내고자 하는 경로의 임의의 이름이다. GLib.Test 클래스는 각 함수가 테스트 경로에 의해 표현되도록 요한다. 그 다음, 루트 경로부터 시작해 모든 경로를 실행하도록 Test.run 함수를 호출한다. 루트 경로는 계층 구조에서 최상위 경로인 루트 디렉터리와 같다.

```
Test.add_func ("/test-isbn", test_isbn);
Test.add_func ("/test-title", test_title);
Test.add_func ("/test-add-author", test_add_author);

Test.run ();

return 0;
```

시도해보기 - 실제 값 확인하기

일부 사례에서는 데이터 구조체로 넣는 실제 값, 즉 Book 클래스의 authors 배열 리스트와 같은 값을 확인할 필요가 있다. test_add_author에서는 배열의 길이만 확인하고 실제 값은 확인하지 않았다. 이러한 확인은 스스로 데이터 구조체를 만들 때 또는 데이터 구조체가 복잡할 때도 필요한데, 가령 데이터의 삽입 후 데이터 순서가 중요한 경우를 예로 들 수 있겠다. 하나 이상의 데이터 집합을 사용하여 확인하는 수도 있다.

이제 우리만의 데이터 구조체가 있다고 가정해보자. 따라서 우리가 입력하는 데이터가 실제로 그렇게 입력되었는지 확인하는 것이 임무다.

테스트 스터빙

코드에서는 비결정적인 행위, 즉 단위 테스팅에 나쁜 행위를 제공할 수 있는 외부 라이브러리를 사용할 때가 종종 있다. 이를 막기 위해 "stubbing"(스터빙) 기법을 사용한다. 이 기법은 함수를 모방하는 새로운 라이브러리와 원본 라이브러리의 API를 생성하는 과정을 수반한다. 우리 단위 테스트에서는 원본 라이브러리 대신 이렇게 새로운 가짜 라이브러리를 이용해 코드에서 사용하는 API의 출력을 제어할 수 있도록 한다.

실행하기 - 스텁 생성하기

이제 제 4장, GNOME 코어 라이브러리 사용하기에서 살펴본 core_settings 프로젝트로 돌아가 보자. 이는 특정 값을 이용해 GSettings를 얻고 설정하는 실험이다.

1. Anjuta로 프로젝트를 열어라.
2. 앞의 활동과 같이 기본구조를 구성하라. 즉, tests 디렉터리를 생성하고, 이를 프로젝트의 최상위 디렉터리에 위치한 configure.ac 와 Makefile.am 으로 넣어라.
3. CoreSettings 클래스를 약간 변경해야 한다. 원본 코드는 CoreSettings 클래스와 동일한 파일에 메인 함수를 갖고 있다. 이 둘을 구분해야 한다.
4. 따라서 우리가 할 일은 src/Makefile.am을 조정하는 것이다. 파일에서 구체적으로 SOURCES 섹션을 수정하라. 파일 내 코드는 다음과 같다.

```
core_settings_SOURCES = \
    core_settings.vala config.vapi
```

위를 아래와 같이 변경해야 한다.

```
core_settings_SOURCES = \
    core_settings.vala main.vala config.vapi
```

5. 그리고 src/core_settings.vala를 아래와 같이 수정하라.

```
using GLib;
public class CoreSettings : Object
{
    Settings settings = null;
    public CoreSettings ()
    {
        settings = new Settings("org.gnome.desktop.background");
    }
    public string get_bg()
    {
        if (settings == null) {
            return null;
        }
        return settings.get_string("picture-uri");
    }
    public void set_bg(string new_file)
    {
        if (settings == null) {
```

```
            return;
        }
        if (settings.set_string ("picture-uri", new_file)) {
            Settings.sync ();
        }
    }
}
```

6. 이후 원본 메인 함수를 포함하는 새로운 파일 src/main.vala를 생성하라. 단, 이번에는 Main 이라 불리는 전용 클래스로 넣어라.

```
public class Main {
    static int main (string[] args)
    {
        var app = new CoreSettings ();
        stdout.printf("%s\n", app.get_bg());
        app.set_bg
("http://www.gnome.org/wp-content/themes/gnome-grass/images/gnome-logo.png");
        return 0;
    }
}
```

7. 나누는 작업이 끝나면 tests 디렉터리로 넘어간다. 먼저 디렉터리에 Makefile.am을 생성하라. 파일에는 아래의 코드를 사용한다.

```
AM_CPPFLAGS = \
    -DPACKAGE_LOCALE_DIR=\""$(localedir)"\" \
    -DPACKAGE_SRC_DIR=\""$(srcdir)"\" \
    -DPACKAGE_DATA_DIR=\""$(pkgdatadir)"\" \
    $(CORE_SETTINGS_CFLAGS)

AM_CFLAGS =\
    -Wall\
    -g

TESTS=test_settings

check_PROGRAMS = test_settings

test_settings_SOURCES = \
    ../src/core_settings.vala test_settings.vala stub/gsettings.vala

test_settings_VALAFLAGS = \
    --pkg gee-1.0

test_settings_LDFLAGS = \
    -Wl,--export-dynamic
```

```
test_settings_LDADD = $(CORE_SETTINGS_LIBS)
```

8. 다음으로 tests 디렉터리에 새로운 파일 test_settings.vala 를 생성하라.

```
public class TestSettings {
    static void test_set_get()
    {
        var s = new CoreSettings();
        s.set_bg("test123");
        assert (s.get_bg() == "test123");
    }

    static int main (string[] args)
    {
        Test.init (ref args);

        Test.add_func ("/test-set-get", test_set_get);

        Test.run ();

        return 0;
    }
}
```

9. 그 다음 tests 디렉터리에 stub 이라는 디렉터리를 생성하라.
10. stub 디렉터리에 새로운 파일 gsettings.vala 를 생성하라. 이 파일은 아래의 코드로 채워라.

```
using Gee;

public class Settings : Object {
    HashMap<string,string> map;

    [CCode(cname="g_settings_new")]
    public Settings(string s) {
        map = new HashMap<string,string>();
    }

    [CCode(cname="g_settings_sync")]
    public static void sync() {
        /* do nothing */
    }

    [CCode(cname="g_settings_set_string")]
    public bool set_string(string key, string value) {
        map.set(key, value);
        return true;
```

```
    }

    [CCode(cname="g_settings_get_string")]
    public string get_string(string key) {
        return map.get(key);
    }
}
```

11. Terminal에서 아래의 명령을 호출하여 프로젝트를 재빌드하라.

```
./autogen.sh
```

12. 마지막으로 아래의 행을 실행하라.

```
make check
```

13. 테스트가 아래와 같은 메시지를 표시하면서 완전하게 통과되도록 확실히 하라.

```
/test-set-get: OK
PASS: test_settings
=============
1 test passed
```

무슨 일이 일어났는가**?**

기본구조를 빌드한 후에는 코드를 두 개의 부분, 즉 CoreSettings 클래스와 Main 클래스로 나누었다. 프로그램에 하나 이상의 main 함수를 가질 수 없기 때문에 이 과정은 꼭 필요하며, 하나는 코드로부터, 나머지 하나는 테스트로부터 비롯된다. 이 과정이 필요한 더 중요한 이유는, 테스트하고자 하는 클래스가 다른 곳에는 속하지 않고 클래스에만 속하는 코드를 포함하기 때문이다. 여기까지 완료하면 깨끗하고 정돈된 코드를 갖게 되며, 무언가 잘못되면 추적하기 수월할 것이다.

이는 원본 메인 함수를 고유의 클래스가 있는 고유의 파일로 넣음으로써 해결할 수 있다. 원본 Main 클래스의 이름도 그 기능을 반영하도록 CoreSettings로 다시 명명한다 (GNOME의 코어 라이브러리에서 Settings를 이용).

그리고 테스트 파일을 tests 디렉터리에 넣는다. tests 디렉터리에 stub이라는 새로운 하위디렉터리도 생성하였다. 그 안에는 gsettings.vala 파일이 있다. 해당 파일은 GSettings 클래스의 이미테이션을 포함한다. 스텁(stub)은 우리가 코드에서 사실상 사용하는 함수만 포함한다. 그 외 사용하지 않는 함수는 구현해선 안 된다.

이번 예제에서는 Gee로부터 해시 맵 구현을 이용해 GSettings를 구현한다.

```
using Gee;
public class Settings : Object {
    HashMap<string,string> map;
```

코드에 생성자가 사용되었으므로 이를 구현할 필요가 있다. 내부에서 해시 맵을 초기화한다. 인자는 중요하지 않으므로 어디에도 보관할 필요가 없다. Vala에서 스터빙은 매우 까다롭다. 이는 Vala 코드가 사용하는 함수들이 사실 생성된 C 함수들이기 때문이다. 따라서 원본 라이브러리를 이용한 스텁에 정확히 동일한 클래스명과 함수명이 있다 하더라도 생성된 클래스 및 함수명은 더 이상 생성된 C 파일에서 동일하지 않다. 스텁에서 클래스명과 함수명의 차이는 테스트 프로그램에서 사용되지 않을 것이다.

이러한 문제를 해결하기 위해서는 스터빙된 라이브러리와 동일한 이름을 생성할 것을 Vala에게 알릴 필요가 있다. 따라서 Vala 코드에서 클래스 선언 또는 함수 선언 직전에 CCode 속성을 이용한다. 본문에 제시한 예제에서는 이 속성 바로 뒤에 생성자가 따라온다.

```
[CCode(cname="g_settings_new")]
public Settings(string s) {
    map = new HashMap<string,string>();
}
```

이는 Vala가 Settings 생성자로부터 g_settings_new C 함수를 생성하도록 확보할 것이다.

코드에서 Settings.sync 함수를 사용하므로 sync 함수를 정의할 필요가 있다. GSettings API 참조에 따라 함수는 정적(static)이므로 그와 동일하게 생성해야 한다.

```
[CCode(cname="g_settings_sync")]
public static void sync() {
    /* do nothing */
}
```

기능이 필요하지 않더라도 함수를 생성해야 한다. 이 과정을 건너뛰면 Vala는 gsettings.vala에서 sync 함수를 찾을 수 없을 것이고 원본 라이브러리로부터 이름을 결정(resolve)하려는 시도를 하지 않을 것이기 때문에 결국 컴파일이 실패할 것이다.

그 다음으로 set_string 함수를 생성한다. 이 함수에서는 단순히 해시 맵을 래핑하고 key와 value 쌍을 맵으로 삽입한다.

```
[CCode(cname="g_settings_set_string")]
public bool set_string(string key, string value) {
    map.set(key, value);
    return true;
}
```

get_string도 마찬가지다. CCode 속성을 사용하고, 요구되는 C 함수명을 정의해야 함을 주목한다.

```
    [CCode(cname="g_settings_get_string")]
    public string get_string(string key) {
        return map.get(key);
    }
}
```

이제 테스트 코드를 한 번 살펴보자.

```
public class TestSettings {
    static void test_set_get()
    {
        var s = new CoreSettings();
        s.set_bg("test123");
        assert (s.get_bg() == "test123");
    }
```

CoreSettings 클래스 코드를 바탕으로 set_bg와 get_bg 함수만 테스트하면 된다. 클래스에서 기능을 제공하는 함수는 더 이상 존재하지 않는다. 따라서 곧바로 main 함수를 구현한다.

```
    static int main (string[] args)
    {
        Test.init (ref args);
        Test.add_func ("/test-set-get", test_set_get);
        Test.run ();
        return 0;
    }
}
```

여기서는 Test 프레임워크를 초기화하고, 테스트 경로를 추가한 다음 테스트를 실행한다. 아 참, 스텁은 어디서 사용되었지? tests/Makefile.am 을 다시 살펴보고 미스터리를 풀어보자. SOURCES 섹션을 훑어보자.

```
test_settings_SOURCES = \
    ../src/core_settings.vala test_settings.vala stub/gsettings.vala
```

이 부분은 Vala가 세 개의 파일을 동시에 컴파일해야 한다고 말한다. 테스트 코드는 GSettings로 어떤 참조도 갖고 있지 않으며, core_settings.vala에 있는 소스만 GSettings로의 참조를 갖고 있다. 따라서 core_settings.vala를 살펴보면 아래의 내용이 보일 것이다.

```
public CoreSettings ()
{
    settings = new Settings("org.gnome.desktop.background");
}
```

이는 사실상 GSettings 대신 gsettings.vala로부터 settings 객체를 인스턴스화하는데, 여기서는 gsettings.vala로부터 get_string을 호출한다.

```
public string get_bg()
{
    if (settings == null) {
        return null;
    }
    return settings.get_string("picture-uri");
}
```

스터빙에서 핵심은 우리가 사용하는 라이브러리와 정확히 똑같은 API를 생성하는 것이다. 빌드를 할 때 코드, 테스트, 스텁을 동시에 컴파일한다. 그렇지 않으면 Vala는 올바르지 않은 API를 사용하고 있다거나 심지어 빌드가 성공적으로 이루어지지 않았다고 불평하여 결국 스텁이 제대로 호출되지 않을 것이다.

GUI 모듈 테스트하기

지금까지는 함수를 간편하게 테스트하여 인자로 전달하는 값을 기반으로 결과를 제공하였다. 하지만 GUI는 사용자 행위로부터 입력, 즉 마우스의 클릭이나 키보드의 타이핑과 같은 입력을 기대한다. 좀 더 일반적으로 말해, GUI는 하나 또는 그 이상의 이벤트를 바탕으로 특정 출력을 제공함으로써 응답한다.

GUI 모듈을 테스트할 때는 앞서 학습한 방법을 더 이상 사용할 수가 없다. GUI 모듈을 테스트할 때는 아래의 측면들이 처리되어야 한다.

- GUI 테스트 애플리케이션을 준비하기 위한 환경 설정
- 그래픽 프레임워크 초기화
- UI 이벤트의 발생과 처리
- 이벤트 루프 관리

이제 이 문제들을 논해보자.

실행하기 - GTK+ 모듈 테스트하기

기존의 custom_composite Vala 프로젝트에 단위 테스트를 넣어보자.

1. 언제나처럼 tests 디렉터리를 생성하여 최상위 디렉터리에 위치한 configure.ac와 Makefile.am으로 넣어보자.
2. 코드를 두 부분으로 나누어라. custom_window.vala에 다음과 같은 코드를 사용하라.

```
using GLib;
using Gtk;

public class CustomWindow : Window
{
    Entry entry;
    Box box;
    public signal void search_updated(string value);

    void show_search_box() {
        entry.show();
        entry.has_focus = true;
    }

    void hide_search_box() {
        entry.hide();
    }
    public override void add(Widget widget) {
        if (widget != box) {
            box.pack_start(widget, true, true);
        } else {
            base.add(widget);
        }
    }

    public CustomWindow ()
    {
        box = new Box(Orientation.VERTICAL, 0);
        entry = new Entry();
```

```
        box.pack_start (entry, false, true);
        box.show();

        add(box);

        key_release_event.connect((event) => {
            search_updated(entry.text);
            return false;
        });

        key_press_event.connect((event) => {
            if (!entry.get_visible()) {
                show_search_box();
            }
            return false;
        });
    }
}
```

3. main 클래스에는 아래의 코드를 사용하라.

```
using GLib;
using Gtk;

public class Main {
    static int main (string[] args)
    {
        Gtk.init (ref args);
        var window = new CustomWindow();
        var label = new Label("This is a text");

        window.add(label);
        window.resize(400,400);
        window.search_updated.connect((value) => {
            label.set_text("Searching for keyword " + value);
        });

        label.show();
        window.show();
        Gtk.main ();

        return 0;
    }
}
```

4. 그리고 구현 클래스와 메인 클래스 모두를 포함하도록 Makefile.am 파일을 수정하라.

```
custom_composite_SOURCES = \
    custom_composite.vala main.vala config.vapi
```

5. 다음으로 테스트용 Makefile.am 을 생성하라.

```
AM_CPPFLAGS = \
    -DPACKAGE_LOCALE_DIR=\""$(localedir)"\" \
    -DPACKAGE_SRC_DIR=\""$(srcdir)"\" \
    -DPACKAGE_DATA_DIR=\""$(pkgdatadir)"\" \
    $(CUSTOM_COMPOSITE_CFLAGS)

AM_CFLAGS =\
    -Wall\
    -g

TESTS=test_custom_window
check_PROGRAMS = test_custom_window

test_custom_window_SOURCES = \
    ../src/custom_composite.vala test_custom_window.vala

test_custom_window_VALAFLAGS = \
    --pkg gtk+-3.0

test_custom_window_LDFLAGS = \
    -Wl,--export-dynamic

test_custom_window_LDADD = $(CUSTOM_COMPOSITE_LIBS)
```

6. tests 안에 새로운 파일 test_custom_window.vala를 생성하고 아래의 코드를 사용해보라.

```
using Gtk;

public class TestCustomWindow {

    static void process_events()
    {
        while (Gtk.events_pending ()) {
            Gtk.main_iteration_do(true);
        }
    }

    static void test_initial_child ()
    {
        var window = new CustomWindow();
        var child = window.get_child () as Box;
        window.show_now();

        assert (child != null);
```

```
        window.destroy ();
    }

    static void test_child_visibility ()
    {
        var window = new CustomWindow();
        var child = window.get_child () as Box;
        window.show_now();

        var entry_is_found = false;
        var children = child.get_children ();
        if (children != null && children.nth(0) != null) {

            var entry = children.nth_data(0) as Entry;

            assert (entry != null);
            assert (entry.visible == false);

            Gdk.test_simulate_key (window.get_window (), 1, 1,
Gdk.Key.a, 0, Gdk.EventType.KEY_PRESS);
            Gdk.test_simulate_key (window.get_window (), 1, 1,
Gdk.Key.a, 0, Gdk.EventType.KEY_RELEASE);

            process_events (); // Process events

            assert (entry.visible == true);

            entry_is_found = true;
        }

        assert (entry_is_found);
        window.destroy ();
    }

    static void test_search_updated ()
    {
        var window = new CustomWindow();
        window.show_now();
        var search_updated_was_emitted = false;
        var search_updated_was_correct = false;

        window.search_updated.connect ((text) => {
            search_updated_was_emitted = true;
            if (text == "a") {
                search_updated_was_correct = true;
            }
        });
```

```
        Gdk.test_simulate_key (window.get_window (), 1, 1, Gdk.Key.a,
0, Gdk.EventType.KEY_PRESS);
        Gdk.test_simulate_key (window.get_window (), 1, 1, Gdk.Key.a,
0, Gdk.EventType.KEY_RELEASE);

        process_events (); // process events

        assert (search_updated_was_emitted);
        assert (search_updated_was_correct);
        window.destroy ();
    }

    static int main (string[] args)
    {
        Gtk.test_init (ref args);

        Test.add_func ("/test-search-updated", test_search_updated);
        Test.add_func ("/test-initial-child", test_initial_child);
        Test.add_func ("/test-child-visibility",
test_child_visibility);

        Idle.add (() => {
            Test.run ();
            Gtk.main_quit ();
            return true;
        });

        Gtk.main ();

        return 0;
    }
}
```

7. 아래 명령을 이용해 프로젝트를 다시 빌드하라.

```
./autogen.sh
```

8. 아래 명령을 발행하여 GUI 초기화를 위한 환경을 만들어라.

```
export DISPLAY=:0
```

   위 명령은 처음으로 GUI 테스트를 실행할 때 한 번만 실행되어야 한다. 이는 terminal 콘솔에서 테스트를 실행할 때 필요하며, 테스트를 GNOME 내부에서 직접 실행할 경우에는 필요 없다.

9. 테스트를 실행하면 테스트가 완료되기 전에 잠시 동안 창이 깜빡임을 확인할 수 있을 것이다. 테스트를 실행하기 위해선 셸에 아래 명령을 입력한다.

```
make check
```

10. 모든 테스트를 성공적으로 통과하는지 확인하라.

```
/test-search-updated: OK
/test-initial-child: OK
/test-child-visibility: OK
PASS: test_custom_window
==============
1 test passed
==============
```

무슨 일이 일어났는가**?**

이제 단위 테스트가 어떻게 이루어지는지에만 집중하자.

CustomWindow 클래스가 무엇을 하는지 살펴보자.

- 이것은 창에 해당하며, 내부에 텍스트 엔트리를 생성한다.
- 창은 하나의 자식만 취할 수 있으며, 자식을 취하더라도 텍스트 엔트리를 제거해선 안 된다.
- 사용자가 창에 있는 키를 누르면 search_updated 시그널이 발생한다.

이는 단위 테스트를 생성하는 전략에서 기본이다.

첫 번째 테스트를 위해 생성자가 Entry 와 다른 위젯의 플레이스홀더로 Box 객체를 올바로 생성하는지 확인한다.

```
static void test_initial_child ()
{
    var window = new CustomWindow();
    var child = window.get_child () as Box;
    window.show_now();
    assert (child != null);
    window.destroy ();
}
```

여기서는 단순히 생성자가 Box 객체를 갖고 있는지 유무를 확인한다. 꽤 간단한 테스트다.

다음 테스트에서는 텍스트 입력의 가시도(visibility)를 확인한다. 처음에 텍스트 엔트리는 숨겨져 있지만 어떤 키든 누르게 되면 엔트리가 보여야 한다.

```
static void test_child_visibility ()
{
    var window = new CustomWindow();
    var child = window.get_child () as Box;
    window.show_now();
```

여기서 창을 즉시 표시하기 위해선 show_now를 호출한다. show를 사용하면 다른 이벤트들이 처리될 때까지 표시가 지연될 수 있다.

```
var entry_is_found = false;
var children = child.get_children ();
if (children != null && children.nth(0) != null) {
```

```
    var entry = children.nth_data(0) as Entry;
```

다음으로 텍스트 엔트리를 얻는다. 엔트리는 창의 첫 번째 자식이므로 nth_data 메서드를 사용하여 색인 0에 위치한 요소를 나타내기 위해 0을 전달한다.

```
assert (entry != null);
assert (entry.visible == false);
```

리턴된 위젯을 Entry 위젯으로 형변환(cast)하기 위해 Entry 를 사용한다. 이러한 형변환을 이용하면 리턴된 위젯이 Entry 위젯이 아닐 경우 null을 리턴할 것이다. 여기서는 entry 변수를 null 값과 비교함으로써 변수가 실제로 Enry 위젯 타입인지 확인한다. 성공하면 가시도가 false로 설정되었는지 계속 확인하는데, 엔트리의 초기 가시도가 false여야 하기 때문이다.

```
Gdk.test_simulate_key (window.get_window (), 1, 1, Gdk.Key.a, 0,
Gdk.EventType.KEY_PRESS);
Gdk.test_simulate_key (window.get_window (), 1, 1, Gdk.Key.a, 0,
Gdk.EventType.KEY_RELEASE);
```

다음으로 Gdk 클래스로부터 test_simulate_key를 사용함으로써 키 누름(key press) 다음에 키 누름해제(key release)의 시뮬레이션을 실행한다. 해당 이벤트들을 window 객체로 전송한다. 함수는 Gdk.Window를 필요로 하므로 get_window를 이용해 window 객체로부터 Gdk.Window를 전달한다. window 객체를 표시하기 위해선 꼭 show_now를 사용해야 하는데, 단순히 show를 사용 시 표시하기가 실행되지 않으면 get_window 함수는 null을 리턴하고 테스트는 실패할 것이기 때문이다.

누름과 누름해제 이벤트의 어떤 위치든 이용할 수 있다. 여기서는 (1,1) 좌표를 사용한다. 우리가 전달하는 키는 a 키이므로 어떤 수식키(key modifier)도 없이 Gdk.Key.a 를 사용한다. 누름 이벤트를 먼저 보낸 다음 누름해제 이벤트를 전송한다.

```
        process_events (); // Process events
        assert (entry.visible == true);

        entry_is_found = true;
    }
    assert (entry_is_found);
    window.destroy ();
}
```

키를 전송한 직후, process_events 함수를 호출함으로써 대기 중인 이벤트를 처리할 필요가 있다. 여기까지 완료되면 이제 가시도를 확인할 수 있다. process_events를 호출하지 않으면 키 이벤트가 시스템에 의해 처리되지 않아 entry 객체의 가시도가 변경되지 않을 수도 있다. 그 다음, 창을 종료(destroy)한다.

process_events 함수가 하는 일을 살펴보자.

```
static void process_events()
{
    while (Gtk.events_pending ()) {
        Gtk.main_iteration_do(true);
    }
}
```

위는 기본적으로 큐에 대기 중인 이벤트는 모두 처리한다. 이벤트를 발견하면 단순히 main_iteration_do를 호출함으로써 메인 루프를 실행하고, 설사 GTK+가 막고 있더라도 우리는 true 인자를 제공함으로써 GTK+가 연산을 완료하도록 허용한다.

다음으로 시그널이 올바른 값으로 적절하게 발생하였는지 확인한다.

```
static void test_search_updated ()
{
    var window = new CustomWindow();
    window.show_now();
    var search_updated_was_emitted = false;
    var search_updated_was_correct = false;

    window.search_updated.connect ((text) => {
        search_updated_was_emitted = true;
        if (text == "a") {
            search_updated_was_correct = true;
        }
    });
```

여기서는 search_updated 시그널을 연결한다. 로컬 변수를 이용해 핸들러의 성공을 기록한다. 그리고 시그널로부터 텍스터가 a 인지 확인하는데, 이는 창으로 전송하는 키 이벤트에 해당한다.

```
Gdk.test_simulate_key (window.get_window (), 1, 1, Gdk.Key.a, 0,
Gdk.EventType.KEY_PRESS);
Gdk.test_simulate_key (window.get_window (), 1, 1, Gdk.Key.a, 0,
Gdk.EventType.KEY_RELEASE);

process_events (); // process events
```

키 이벤트를 전송하고 나면 process_events를 호출한다. 그 이유는 시그널 핸들러가 호출되어 결과를 주장할 수 있도록 확보하기 위함이다.

```
assert (search_updated_was_emitted);
assert (search_updated_was_correct);
window.destroy ();
```

main 함수에서는 먼저 Gtk.test_init 함수를 호출한다. 이는 테스트에 적합한 환경을 구성하기 위함이다. 다음으로 테스트 경로를 추가한다.

```
static int main (string[] args)
{
    Gtk.test_init (ref args);

    Test.add_func ("/test-search-updated", test_search_updated);
    Test.add_func ("/test-initial-child", test_initial_child);
    Test.add_func ("/test-child-visibility", test_child_visibility);
```

GTK+ 시스템을 준비하여 실행시키기 위해선 여전히 Gtk.main이 필요하다. 테스트를 GNOME 밖에서 실행할 때는 테스트가 실행될 수 있도록 X11 서버를 실행하고 환경을 구성할 필요가 있다. 이는 DISPLAY 환경 변수를 우리 X11 표시 번호로 설정함으로써 실행된다. 보통 값은 :0이다. 원격 테스트를 실행할 경우 약간 번거로운 셋업에 해당한다.

하지만 우선 실행되고 나면 테스트가 실행되는지 확보할 필요가 있다. 따라서 idle 핸들러를 구성할 필요가 있다. 핸들러 내부에서는 간단히 테스트를 실행하고 난 후 GTK+ 시스템을 종료한다.

```
Idle.add (() => {
    Test.run ();
    Gtk.main_quit ();
    return true;
});

Gtk.main ();
```

시도해보기 - 빠진 테스트 추가하기

앞의 테스트에서는 window 객체 내부에서 Box 객체의 존재만 확인하였다. 따라서 확인해야 할 또 다른 필수 테스트를 빠뜨린 셈이다.

- Entry가 실제로 생성되었는지 여부
- 새로운 위젯을 window 객체로 추가할 때 Entry가 동시에 존재할 수 있는지 여부

답을 알아내고 테스트를 생성해보자!

요약

GNOME 코드에서 단위 테스팅을 어떻게 실행하는지 살펴보았다. 초보자에게 가장 까다로운 부분은 어떤 테스트를 써야 하는지 알아내는 것이다. 이 주제를 논하기 위해서는 먼저 테스트하길 원하는 코드가 제공하는 기능을 먼저 확인해야 한다. 그 다음 다른 인자를 제공함으로써 코드에 존재하는 각 branch로 좁혀나가면 모든 사례에서 코드 경로를 살펴볼 수 있을 것이다.

결정적인 테스트를 생성해야 하지만 우리가 사용하는 라이브러리는 (심지어 본인의 코드조차) 비결정적인 행위를 제공할 수 있음을 이해했을 것이다. 이러한 문제를 해결하기 위해서는 각 라이브러리에 대해 잠재적으로 비결정적 값을 제공할 수 있는 스텁을 생성할 필요가 있다. 사용 중인 라이브러리가 너무 복잡하거나 단위 테스트를 실행할 때 복잡한 셋업을 필요로 하는 경우에도 스터빙을 실행할 수 있다.

마지막으로, GTK+ 고유의 초기화 함수를 이용함으로써 GUI 모듈을 테스트하는 방법을 학습하였다. 무언가를 주장하기 전에는 대기 중인 이벤트를 모두 처리할 필요가 있다. 또 즉시 Gdk 창을 구성하기 위해서는 show_now 함수를 사용해야 함을 배웠다.

우리가 쓴 테스트는 코드를 변경할 때마다 실행되어야 한다. 그 이유는 코드에서 회귀(regression)를 원치 않기 때문이다. 많은 소프트웨어 개발자들이 소스 코드 저장소로 코드를 전송하기 전에 테스트를 실행하고, 존재하는 모든 테스트 사례를 실행함으로써 밤마다 테스트를 구성한다.

마지막 장에서는 두 가지 큰 프로젝트, 브라우저와 트위터(twitter) 클라이언트를 실행하는 방법을 학습할 것이다. 지금까지 학습한 기법을 모두 사용할 예정이니 마음을 다잡도록!

Notes

GNOME3ApplicationDevelopmentBeginnersGuide:Chap 13

제 13 장 흥미로운 프로젝트

흥미로운 프로젝트

별로 유용하지 않은 애플리케이션을 개발하는 방법을 학습하는 데에 많은 시간을 소요했으니 이제 다음 단계로 넘어갈 차례다. 이제 웹 브라우저와 Twitter 클라이언트를 개발해보도록 하겠다. 우리가 만들고자 하는 애플리케이션은 간단하면서도 유용하다. 고급 버전을 만들기로 결정했다면 기본으로 사용해도 좋을 내용이다.

이번 장에서는 지금까지 배운 대부분의 주제들을 다시 살펴보되 약간 복잡하게 만들어 좀 더 깊이 알아보고자 한다. 구체적으로 익히게 될 지식은 다음과 같다.

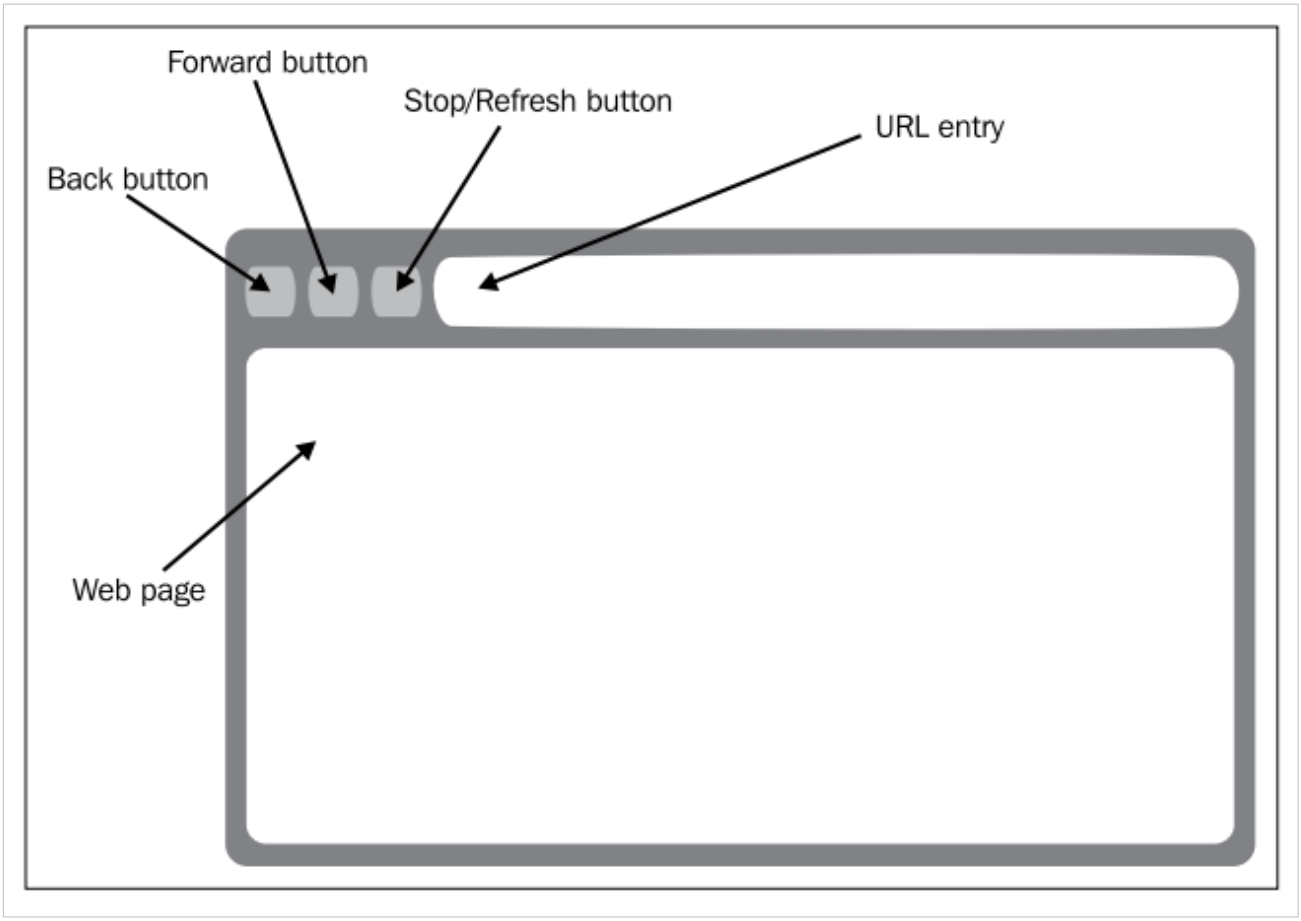
- WebKitGTK를 이용해 웹 브라우저 구현하기
- Configure 스크립트를 이용해 경로 하드코딩하기
- 다중 스크립트 Seed 애플리케이션 개발하기

첫 프로젝트부터 시작해보자.

제 1부 - 웹 브라우저

제일 먼저 개발할 애플리케이션은 웹 브라우저다. 개발이 단순한 기본 브라우저를 목표로 하겠다. 브라우저에는 탐색 버튼의 세트와 인터넷으로부터 페이지를 여는 기능이 포함되어 있다.

전체적인 개발이 수월해지도록 브라우저의 목업 버전을 만든다. 목업은 웹 브라우저의 모습이 어떨지를 보여주는 그림에 불과하다. 이러한 목업을 바탕으로 Anjuta/Glade 툴에서 UI 레이아웃을 도출할 것이다.



먼저 브라우저의 UI 구조에 익숙해지자. 메인 창은 두 가지 부분으로 나뉜다. 웹 페이지를 표시하는 영역이 창의 대부분 영역을 차지한다. 최상위 부분은 탐색 버튼과 URL 엔트리가 사용한다.

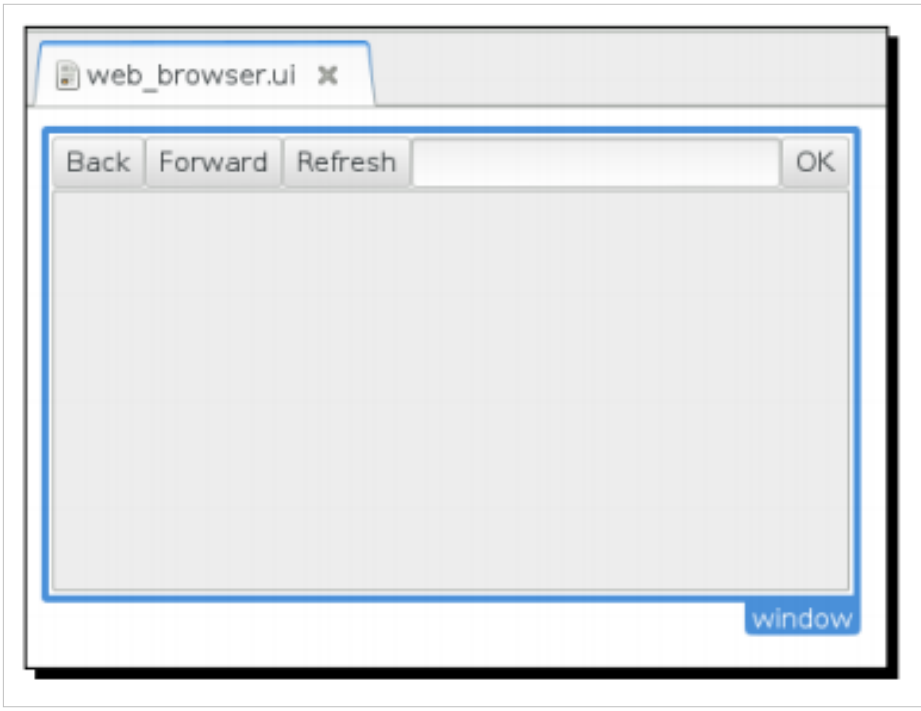
탐색 버튼은 독립된 버튼으로서 각각 액션을 뒤로 이동(back), 앞으로 이동(forward), 중지하기(stop)/재로딩하기(reload)를 나타낸다. URL 엔트리 옆에는 go 버튼이 추가로 숨겨져 있다. 버튼은 URL 엔트리를 채우면 표시된다.

이게 전부다. 정말 간단하다. 그건 그렇고 탐색 버튼과 URL 엔트리를 관리(host)하는 UI는 GUI 디자인 용어로 "chrome"(크롬)이라 부른다.

실행하기 - UI 디자인하기

이제 Anjuta를 작동시켜 UI를 디자인해보자. Glade 디자이너는 팔레트에 GtkWebUI를 갖고 있지 않기 때문에 다음과 같은 작업을 실행해야 한다.

1. 새로운 Vala 프로젝트를 생성하고 web-browser라고 불러라. UI에 GtkBuilder를 사용하라.
2. **Project** 도크에 파일을 더블 클릭하여 Glade 디자이너로 web_browser.ui를 열어라.
3. 두 개의 항목이 있는 창에 수직 Box 객체를 넣어라.
4. 이번에는 수평 Box 박스를 수직 Box 박스의 상단에 놓고, 하단 부분은 빈 채로 두어라. 수평 Box 객체에서 **Number of elements** 옵션을 5로 하여 5개 요소로 나누어라.
5. 세 번째 요소를 제외하고 비어 있는 박스 요소 각각에 **button** 을 넣어라. 각 버튼마다 **Stock button** 옵션을 활성화하고, 각 **button** 에서 gtk-go-back, gtk-go-forward, gtk-refresh, gtk-ok stock으로 채워라.
6. 버튼에 btn_back, btn_forward, btn_refresh, btn_go를 이용해 이름을 제공하라. 이름은 버튼의 각 **Name** 옵션에 넣어라.
7. 세 번째 빈 **box** 에 Entry 객체를 놓고 url_entry로 이름을 붙여라. **Packing** 탭으로 가서 **Expand** 옵션이 **Yes** 로 설정되도록 하라.
8. **Scrolled Window** 객체(**Containers** 섹션에 위치)를 하단의 빈 박스로 넣어라. **Expand** 옵션이 **Yes** 가 되도록 하라.
9. **Horizontal Scrollbar Policy** 와 **Vertical Scrollbar Policy** 값을 **Never** 로 설정하라.
10. UI 파일이 아래와 같은 모습으로 준비되었을 것이다.



무슨 일이 일어났는가**?**

지금까지는 웹 브라우저를 위한 UI 레이아웃을 생성하였다. 볼 수 있듯이 목업으로부터 간단한 해석에 해당한다. 한 가지 빠진 내용이 있는데, 웹 페이지 영역이 여전히 비어 있다는 점이다. 하지만 나중에는 WebView 객체에 대한 컨테이너로 ScrolledWindow를 갖게 된다. ScrolledWindow 컨테이너를 사용하는 목적은 창의 크기가 내용에 따라 조정되지 않도록 확보하는 것인데, 크기 조정은 이미 ScrolledWindow로 넘어갔기 때문이다. WebView는 스스로 스크롤바를 이미 처리하므로 수직축과 수평축에 있는 스크롤바는 비활성화한다.

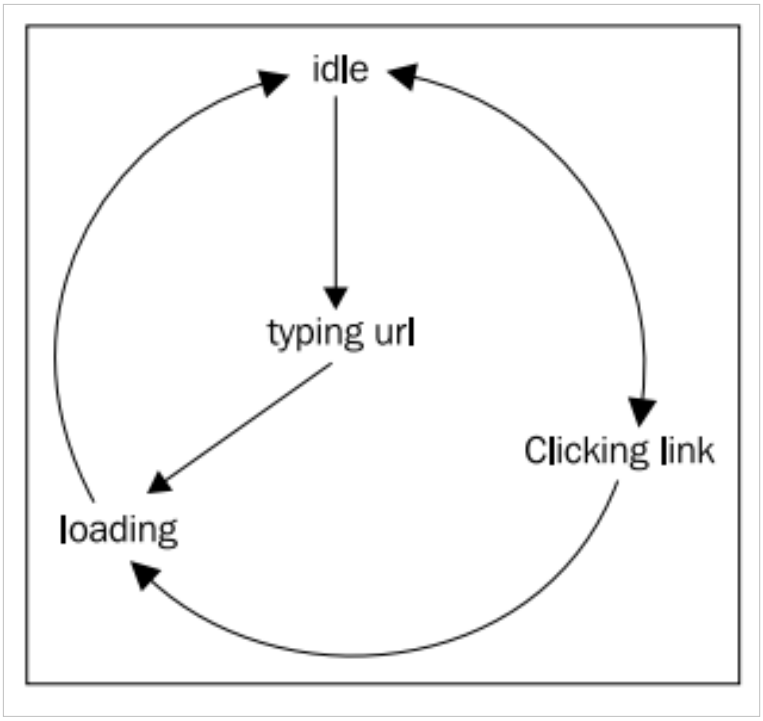
브라우저 상호작용

이제 브라우저의 상호작용 디자인을 검사해보자. 초기 단계에서는 idle 브라우저가 있어서 모든 탐색 버튼이 비활성화되어 있고 URL 엔트리가 비어 있다. 이 때 웹 뷰는 **Welcome** 텍스트만 포함한다 (만일 다른 언어로 지역화되었다면 Welcome에 대한 번역어만 포함한다). 사용자가 그 다음으로 실행할 수 있는 것은 URL 엔트리를 채우는 일이다.

사용자가 무언가를 입력하기 시작하면 go 버튼이 나타난다. 아직 표시할 내용이 없기 때문에 모든 탐색 버튼은 비활성화되어 있다. 따라서 사용자가 입력을 계속하거나 go 버튼을 클릭하여 프로세스를 완료할 것을 결정한다. 둘 중 하나라도 발생하면 웹 뷰는 요청된 URL을 로딩하기 시작한다.

페이지가 로딩되는 동안 refresh 버튼은 스스로를 stop 버튼으로 변경한다. 사용자가 이 버튼을 누르면 로딩이 즉시 중단되고 모든 것을 취소한다. stop 버튼은 refresh 버튼으로 다시 변경된다. 어떤 상태든 브라우저는 유휴(idle) 상태로 유지된다.

사용자가 로딩된 웹 페이지와 상호작용을 계속할 때마다 탐색 버튼의 상태는 그에 따라 업데이트된다. 브라우저가 뒤로 돌아갈 수 있다면 back 버튼이 활성화되었음을 의미한다. forward 버튼도 마찬가지다.



앞의 설명을 바탕으로 하면 네 개의 상태가 브라우저로 적용되어 있다. 사용자가 상호작용을 시작하길 기다리는 동안에는 "idle"(유휴) 상태다. 이후 사용자가 무언가를 입력하기 시작하면 "typing url"(url 입력) 상태다. 아니면 사용자가 링크나 페이지를 클릭할 때마다 "Clicking link"(링크 클릭) 상태로 들어갈 가능성도 있다. 이후 어떤 페이지가 로딩되고 있다면 상태는 "loading"(로딩) 상태가 된다. 마지막에는 다시 유휴 상태가 될 것이다.

이제 이러한 디자인을 사용해보았으니 코드를 구현할 준비가 된 셈이다.

실행하기 - 빌드 기본구조 준비하기

이제 앞서 생성한 프로젝트를 수정하여 **i18n(**국제화**)** 으로 개선한 후 빌드 기본구조를 테스트해보자.

1. configure.ac 파일을 수정하여 아래와 같은 모습이 되도록 하라.

```
AC_INIT(web_browser, 0.1)
AC_CONFIG_HEADERS([config.h])
AM_INIT_AUTOMAKE([1.11])
AM_SILENT_RULES([yes])
AC_PROG_CC

LT_INIT
IT_PROG_INTLTOOL()
AH_TEMPLATE([GETTEXT_PACKAGE], [Package name for gettext])
GETTEXT_PACKAGE=web-browser

AC_DEFINE_UNQUOTED([GETTEXT_PACKAGE], ["$GETTEXT_PACKAGE"],
            [The domain to use with gettext])
AC_SUBST(GETTEXT_PACKAGE)
AM_GLIB_GNU_GETTEXT

dnl Check for vala
AM_PROG_VALAC([0.10.0])
```

```
dnl Development mode
AC_ARG_ENABLE(development,
    AS_HELP_STRING([--enable-development],[enable development mode]),
        enable_development="$enableval",
            enable_development=no)
if test "x$enable_development" = "xyes"; then
    DEVELOPMENT_MODE="yes"
    PACKAGE_LOCALE_DIR=[${PWD}/locale]
    PACKAGE_UI_DIR=[${PWD}/src]
else
    PACKAGE_LOCALE_DIR=[${datadir}/locale]
    PACKAGE_UI_DIR=[${datadir}/web-browser]
fi

AC_SUBST(PACKAGE_LOCALE_DIR)
AC_SUBST(PACKAGE_UI_DIR)
AH_TEMPLATE([PACKAGE_UI_DIR], [Location of the .ui file])
AC_DEFINE_UNQUOTED([PACKAGE_UI_DIR], ["$PACKAGE_UI_DIR"],
            [Location of the .ui file])

AC_SUBST(DEVELOPMENT_MODE)
AH_TEMPLATE([DEVELOPMENT_MODE], [Whether in development mode or not])
AC_DEFINE_UNQUOTED([DEVELOPMENT_MODE], ["$DEVELOPMENT_MODE"],
            [Development mode])

PKG_CHECK_MODULES(WEB_BROWSER, [gtk+-3.0 webkitgtk-3.0])
PKG_CHECK_MODULES(TEST_WEB_BROWSER, [gtk+-3.0 ])

AC_OUTPUT([
Makefile
src/Makefile
tests/Makefile
po/Makefile.in
])
```

2. Makefile.am 파일을 수정하여 아래의 코드로 채워라.

```
SUBDIRS = src tests po

web_browserdocdir = ${prefix}/doc/web_browser
web_browserdoc_DATA = \
    README\
    COPYING\
    AUTHORS\
    ChangeLog\
    INSTALL\
```

```
    NEWS

EXTRA_DIST = \
    $(web_browserdoc_DATA)
    intltool-extract.in \
    intltool-merge.in \
    intltool-update.in

DISTCLEANFILES = \
    intltool-extract \
    intltool-merge \
    intltool-update \
    po/.intltool-merge-cache \
    $(NULL)

# Remove doc directory on uninstall
uninstall-local:
    -rm -r $(web_browserdocdir)
```

3. src/Makefile.am 파일을 수정하고 아래의 전체 파일을 사용하라.

```
uidir = $(PACKAGE_UI_DIR)
ui_DATA = web_browser.ui

AM_CPPFLAGS = \
    -DPACKAGE_LOCALE_DIR=\""$(localedir)"\" \
    -DPACKAGE_SRC_DIR=\""$(srcdir)"\" \
    -DPACKAGE_DATA_DIR=\""$(pkgdatadir)"\" \
    $(WEB_BROWSER_CFLAGS)

AM_CFLAGS =\
    -Wall\
    -g

bin_PROGRAMS = web_browser
web_browser_SOURCES = \
    main.vala web_browser.vala config.vapi

web_browser_VALAFLAGS = \
    --vapidir . \
    --pkg gtk+-3.0 \
    --pkg webkit-1.0 \
    --pkg libsoup-2.4 \
    --Xcc='--include config.h'

web_browser_LDFLAGS = \
    -Wl,--export-dynamic
```

```
web_browser_LDADD = $(WEB_BROWSER_LIBS)

EXTRA_DIST = $(ui_DATA)

# Remove ui directory on uninstall
uninstall-local:
    -rm -r $(uidir)
    -rm -r $(pkgdatadir)
```

4. po 디렉터리를 생성하라.
5. 아래의 내용으로 된 po/POTFILES.in을 추가하라.

```
[type: gettext/glade]src/web_browser.ui
src/web_browser.vala
```

6. 아래의 내용으로 된 po/POTFILES.skip을 추가하라.

```
src/web_browser.c
tests/web_browser.c
```

7. po/LINGUAS 파일을 추가하고 지원하고자 하는 언어 코드로 채워라. 가령 인도네시아어를 지원하기 위해서는 아래의 내용을 넣어라.

```
id
```

8. po 디렉터리 안에서 아래의 명령을 실행하여 번역 템플릿을 채워라.

```
intltool-update -pot
```

9. 결과가 되는 web-browser.pot을 id.po(또는 LINGUAS 파일의 내용에 따라 다른 파일로)로 복사하고 제 11장, 애플리케이션 국제화하기에서 po 파일을 어떻게 처리할 것인지를 참조하라.
10. tests 디렉터리를 생성하라.
11. 아래의 내용을 이용해 tests/Makefile.am을 추가하라.

```
AM_CPPFLAGS = \
    -DPACKAGE_LOCALE_DIR=\""$(localedir)"\" \
    -DPACKAGE_SRC_DIR=\""$(srcdir)"\" \
    -DPACKAGE_DATA_DIR=\""$(pkgdatadir)"\" \
    -I. \
    $(WEB_BROWSER_CFLAGS)

AM_CFLAGS =\
    -Wall\
    -g

TESTS=test_web_browser
```

```
check_PROGRAMS = test_web_browser

test_web_browser_SOURCES = \
    ../src/config.vapi webkit.vala ../src/web_browser.vala test_web_
browser.vala

test_web_browser_VALAFLAGS = \
    --pkg gtk+-3.0 \
    --Xcc='--include config.h'

test_web_browser_LDFLAGS = \
    -Wl,--export-dynamic

test_web_browser_LDADD = $(TEST_WEB_BROWSER_LIBS)
```

12. tests/test_web_browser.vala를 추가하고 아래의 코드로 채워라.

```
using Gtk;

public class TestWebBrowser {

    static void process_events()
    {
        while (Gtk.events_pending ()) {
            Gtk.main_iteration_do(true);
        }
    }
    static int main (string[] args)
    {
        Gtk.test_init (ref args);

        Idle.add (() => {
            Test.run ();
            Gtk.main_quit ();
            return true;
        });

        Gtk.main ();

        return 0;
    }
}
```

13. tests/webkit.vala를 추가하고 아래의 내용을 사용하라.

```
using Gtk;

[CCode (lower_case_cprefix = "webkit_")]
```

```
namespace WebKit {

    [CCode (cheader_filename="webkit/webkit.h")]
    public class WebFrame : Object {
    }

    [CCode (cheader_filename="webkit/webkit.h")]
    public class WebView : Viewport {

        public WebView() {
        }

        [CCode (cname="webkit_web_view_load_string")]
        public void load_string (string content, string mime, string
encoding, string base_uri) {
        }
    }
}
```

14. 본 서적에 함께 따라오는 webkit-1.0.vapi 파일을 src 디렉터리로 넣어라.
15. src/web_browser.vala 파일에 아래 코드를 사용하라.

```
using GLib;
using Gtk;
using WebKit;

public class WebBrowser : Object
{
    internal Builder builder = null;
    WebView view = null;
    const string UI_FILE = Config.PACKAGE_UI_DIR + "/" +
"web_browser.ui";

    public WebBrowser ()
    {
        Gtk.Settings.get_default ().gtk_button_images = true;
        try
        {
            builder = new Builder ();
            builder.add_from_file (UI_FILE);

            view = new WebView();
            view.load_string("<h1>" + _("Welcome") + "</h1>", "text/ html",
"UTF-8", "/");
            var box = builder.get_object ("webhost") as Container;
            box.add(view);
```

```
            var window = builder.get_object ("window") as Window;
            window.show_all ();

            window.destroy.connect(() => {
                Gtk.main_quit();
            });
        }
        catch (Error e) {
            stderr.printf (_("Could not load UI: %s\n"), e.message);
        }
    }
}
```

16. src/main.vala를 생성하고 아래의 내용을 사용하라.

```
using GLib;
using Gtk;

public clas Main : Object
{
    static int main (string[] args)
    {
        Gtk.init (ref args);
        var app = new WebBrowser ();
        Gtk.main ();
        return 0;
    }
}
```

17. src/config.vapi를 아래와 같이 수정하라.

```
[CCode (cprefix = "", lower_case_cprefix = "", cheader_filename =
"config.h")]
namespace Config {
    public const string DEVELOPMENT_MODE;
    public const string GETTEXT_PACKAGE;
    public const string SPRITE_DIR;
    public const string BACKGROUND_DIR;
    public const string PACKAGE_DATA_DIR;
    public const string PACKAGE_UI_DIR;
    public const string PACKAGE_LOCALE_DIR;
    public const string PACKAGE_NAME;
    public const string PACKAGE_VERSION;
    public const string VERSION;
}
```

18. 아래를 실행함으로써 --enable-development 옵션의 사용을 빌드하라.

```
./autogen.sh --enable-development
```

19. 아니면 옵션을 **Build** 메뉴의 **Configure Project...** 하위메뉴에서 **Configure Options** 필드에 넣어라.
20. 콘솔로 가서 아래 명령을 입력하라.

```
make check
```

21. 단위 테스트의 통과 여부를 확인하기 위해서는 이 출력을 확인해야 한다.
22. 애플리케이션을 실행할 수 있어야 하며, 닫는 기능을 제외하고는 어떠한 상호작용도 없는 UI가 표시되어야 한다.

무슨 일이 일어났는가**?**

우아! 꽤 많은 액션이 실행되었다. 그렇지 않은가?

이제 i18n과 단위 테스팅을 포함하는 빌드 기본구조를 준비하였다. 기능은 구현하지 않은 채로 두었다. 구현 또한 빌드 과정이 성공할 정도로만 최소한으로 유지하였다. 이제 좀 더 자세히 살펴보자.

제일 먼저 configure.ac 파일을 구성하였다. 이는 앞의 여러 장에서 실행한 것과 꽤 비슷하지만 몇 가지 흥미로운 점이 있다. 첫 번째는 다음과 같다.

```
dnl Development mode
AC_ARG_ENABLE(development,
    AS_HELP_STRING([--enable-development],[enable development mode]),
        enable_development="$enableval",
            enable_development=no)
if test "x$enable_development" = "xyes"; then
    DEVELOPMENT_MODE="yes"
    PACKAGE_LOCALE_DIR=[${PWD}/locale]
    PACKAGE_UI_DIR=[${PWD}/src]
else
    PACKAGE_LOCALE_DIR=[${datadir}/locale]
    PACKAGE_UI_DIR=[${datadir}/web-browser]
fi
```

위는 configure 스크립트에 --enable-development 옵션의 추가를 보여준다. PACKAGE_LOCALE_DIR와 PACKAGE_UI_DIR의 정의를 추가한 것만 제외하면 꽤 익숙한 모습이다. 이러한 변수들의 값은 configure 스크립트의 매개변수로부터 --enable-development 옵션을 빼는지 추가하는지에 따라 달려있다. PACKAGE_LOCALE_DIR는 제 11장, 애플리케이션 국제화하기에서 논했듯이 LOCALE 디렉터리의 위치를 보유한다. PACKAGE_UI_DIR는 .ui 파일의 위치를 보유한다.

이는 .ui 와 번역 파일을 어디서 찾아야 하는지 코드에게 일일이 알려주는 기능을 보였던 이전 접근법에 비하면 상당히 개선된 모습이다. 주요 개선점으로는 개발 모드에 있는지 혹은 소스 코드에 있지 않은지를 검사할 필요가 없다는 점이다. 위치는 configure 스크립트로 --enable-development 옵션을 제공하는지 여부에 따라 자동으로 하드코딩된다.

옵션을 제공했다면 PACKAGE_UI_DIR 는 src 디렉터리를 가리키고, 제공하지 않았다면 PACKAGE_UI_DIR 는 /usr/share/web-browser 디렉터리를 가리킨다. 이는 컴파일 시 이루어져야 하므로 배포 버전(deployment version)을 빌드하기 전에 --enable-development 옵션을 생략할 것을 잊지 말라.

코드로부터 접근 가능한 PACKAGE_UI_DIR의 값을 얻기 위해서는 값을 config.h 파일로 삽입한다. 이는 configure 스크립트에서 다음과 같이 이루어진다.

```
AC_SUBST(PACKAGE_UI_DIR)
AH_TEMPLATE([PACKAGE_UI_DIR], [Location of the .ui file])
AC_DEFINE_UNQUOTED([PACKAGE_UI_DIR], ["$PACKAGE_UI_DIR"], [Location of
the .ui file])
```

이후 이를 src/config.vapi 파일로 추가함으로써 Vala 코드로부터 접근 가능하게 만들 필요가 있다.

```
public const string PACKAGE_UI_DIR;
```

이후 변수는 Config.PACKAGE_UI_DIR로서 사용된다.

다음으로 아래의 행이 흥미롭다.

```
PKG_CHECK_MODULES(WEB_BROWSER, [gtk+-3.0 webkitgtk-3.0])
PKG_CHECK_MODULES(TEST_WEB_BROWSER, [gtk+-3.0 ])
```

첫 행은 WEB_BROWSER_CFLAGS와 WEB_BROWSER_LIBS의 새 변수를 자동으로 내보내고(export), 두 번째 행은 TEST_WEB_BROWSER_CFLAGS와 TEST_WEB_BROWSER_LIBS 변수를 내보낼 것이다. 애플리케이션을 컴파일하는 데에는 아래와 같이 src/Makefile.am의 첫 번째 변수와,

```
SER, [gtk+-3.0 ])

AM_CPPFLAGS = \
    -DPACKAGE_LOCALE_DIR=\""$(localedir)"\" \
    -DPACKAGE_SRC_DIR=\""$(srcdir)"\" \
    -DPACKAGE_DATA_DIR=\""$(pkgdatadir)"\" \
    $(WEB_BROWSER_CFLAGS)
```

아래를 사용한다.

```
web_browser_LDADD = $(WEB_BROWSER_LIBS)
```

이는 빌드 시스템이 컴파일에 필요한 플래그와 헤더 파일뿐만 아니라 애플리케이션의 연계에 필요한 라이브러리명까지 얻는다는 뜻이다. 이 모든 것은 스텁 컴파일을 성공적으로 만드는 데에 필요하다.

단위 테스팅 실행을 수월하게 만들기 위해 WebKit을 스터빙하는 webkit.vala 코드가 있다. tests/Makefile.am에서 볼 수 있듯이 TEST_WEB_BROWSER_LIBS를 이용해 테스트 파일을 컴파일하면 단위 테스트로 연결된 원본 WebKit 라이브러리를 제외시킬 수 있다. 그러면 src에 포함된 소스 코드는 WebKit로 컴파일될 것이고, tests 디렉터리 내의 소스 코드는 스텁으로 컴파일될 것이다.

하지만 단위 테스트에서는 여전히 TEST_WEB_BROWSER_CFLAGS 대신 WEB_BROWSER_CFLAGS를 이용해 원본 WebKit 헤더 파일을 사용한다. 원본 헤더는 생성된 C 소스 코드를 컴파일하는 데 필요하다.

이제 i18n 부분으로 넘어가보자.

우리의 Vala 코드, 즉 src/web_browser.vala와 .ui 파일을 po/POTFILES.in에 넣음으로써 번역 가능한 파일로 표현해보자. 모든 파일을 추가할 필요는 없으며, 번역 가능한 텍스트가 포함된 파일을 신중하게 선택한다. 그 다음, 모든 생성된 C 코드를 po/POTFILES.skip으로 추가하여 번역 불가한 파일로 표시한다. 이는 이중 번역을 피하기 위함이다.

나머지는 꽤 간단하므로 바로 살펴보겠다.

실행하기 - 끝마치기

1. tests/test_web_browser.vala를 아래의 코드로 채워라.

```
using Gtk;

public class TestWebBrowser {

    static void process_events()
    {
        while (Gtk.events_pending ()) {
            Gtk.main_iteration_do(true);
        }
    }

    static void test_initial_state ()
    {
        var web = new WebBrowser();
        process_events();
        assert (web.state == WebBrowser.State.IDLE);
    }

    static void test_typing_url ()
    {
        var web = new WebBrowser();
        var window = web.builder.get_object ("window") as Window;
        window.show_now ();
        web.url_entry.show_now ();

        assert (web.btn_go.visible == false);
```

```
        var entry_w = web.url_entry.get_window ();
        web.url_entry.focus(0);
        assert (web.state == WebBrowser.State.IDLE);
        Gdk.test_simulate_key (entry_w, 5, 5, Gdk.Key.a, 0,
Gdk.EventType.KEY_PRESS);
        Gdk.test_simulate_key (entry_w, 5, 5, Gdk.Key.a, 0,
Gdk.EventType.KEY_RELEASE);

        process_events ();
        assert (web.state == WebBrowser.State.TYPING_URL);
        assert (web.btn_go.visible == true);

        var btn_w = web.btn_go.get_window ();
        web.btn_go.focus(0);
        Gdk.test_simulate_key (btn_w, 5, 5, Gdk.Key.Return, 0,
Gdk.EventType.KEY_PRESS);
        Gdk.test_simulate_key (btn_w, 5, 5, Gdk.Key.Return, 0,
Gdk.EventType.KEY_RELEASE);

        process_events ();
        assert (web.state == WebBrowser.State.LOADING);
    }
    static int main (string[] args)
    {
        Gtk.test_init (ref args);

        Test.add_func ("/test-initial-state", test_initial_state);
        Test.add_func ("/test-typing-url", test_typing_url);

        Idle.add (() => {
            Test.run ();
            Gtk.main_quit ();
            return true;
        });

        Gtk.main ();

        return 0;
    }
}
```

2. tests/webkit.vala 스텁 파일을 열고 아래 코드를 사용하라.

```
using Gtk;

[CCode (lower_case_cprefix = "webkit_")]
namespace WebKit {
```

```
    [CCode (cheader_filename="webkit/webkit.h")]
    public class WebFrame : Object {
    }

    [CCode (cheader_filename="webkit/webkit.h")]
    public class WebView : Viewport {
        public bool _can_go_back;
        public bool _can_go_forward;
        public signal void load_started (WebFrame frame);
        public signal void load_finished (WebFrame frame);
        WebFrame frame;

        public WebView() {
            frame = new WebFrame ();
        }

        [CCode (cname="webkit_web_view_can_go_back")]
        public bool can_go_back() {
            return _can_go_back;
        }

        [CCode (cname="webkit_web_view_can_go_forward")]
        public bool can_go_forward() {
            return _can_go_forward;
        }

        [CCode (cname="webkit_web_view_go_back")]
        public void go_back () {
        }

        [CCode (cname="webkit_web_view_go_forward")]
        public void go_forward () {
        }

        [CCode (cname="webkit_web_view_load_uri")]
        public void load_uri (string uri) {
            load_started (frame);
        }

        [CCode (cname="webkit_web_view_load_string")]
        public void load_string (string content, string mime, string
encoding, string base_uri) {
        }

        [CCode (cname="webkit_web_view_stop_loading")]
        public void stop_loading () {
        }
```

```
        [CCode (cname="webkit_web_view_reload")]
        public void reload () {
        }
    }
}
```

3. 애플리케이션을 빌드하여 실행하라.

무슨 일이 일어났는가**?**

기본구조를 구성해 놓았으니 이제 프로젝트의 진짜 목적에 집중할 수 있겠다. WebBrowser 객체로 시작해보자.

```
const string UI_FILE = Config.PACKAGE_UI_DIR + "/" + "web_browser.ui";
```

여기서는 파일명 뒤에 붙은 Config.PACKAGE_UI_DIR 값을 얻음으로써 .ui 파일의 정확한 위치를 얻는다. 위치는 이미 빌드 시스템에서 하드코딩되기 때문에 제 11장, 애플리케이션 국제화하기에서처럼 개발 모드인지 아닌지 확인하기 위해 추가로 if branch 코드를 넣을 필요가 없다.

```
internal Button btn_back = null;
internal Button btn_forward = null;
internal Button btn_go = null;
internal Button btn_refresh = null;
internal Entry url_entry = null;
```

이는 탐색 버튼과 URL 엔트리 위젯이다. 단위 테스트로부터 접근이 가능하도록 internal(내부적)로 표시한다.

```
internal enum State {
    IDLE,
    TYPING_URL,
    LOADING
}
```

이는 앞서 논한 상태들이다. 하지만 링크 클릭(clicking link) 상태가 빠져 있다. 클릭은 WebKit 안에서 직접 처리되기 때문에 enum 루프에서 상태를 제거하였고, 그 외에는 우리가 처리하는 상태들이다. 따라서 클러터링(cluttering)을 피하기 위해서는 사용하지 않을 enum 값은 넣지 않을 것이다.

하지만 값을 그냥 제거하는 것은 올바른 실습이 아니다. 제거의 이유를 설명하는 주석을 코드에 추가시키는 편이 낫다.

먼저 초기 상태를 유휴 상태로 초기화한다.

```
internal State state = State.IDLE;
```

이후 상태가 변경될 때마다 호출될 함수가 생긴다. 이 함수에서는 영향을 받는 구성요소의 모양이나 다른 프로퍼티를 업데이트할 것이다.

```
void update_state()
```

아래 코드는 builder 객체가 존재하도록 확보하는 데에 사용된다.

```
if (builder == null) {
    return;
}
```

유휴 상태에서는 go 버튼을 숨기고, refresh 버튼의 아이콘과 라벨을 사용한다.

```
switch (state) {
    case State.IDLE:
        btn_go.hide ();
        btn_refresh.label = "gtk-refresh";
        break;
```

URL을 입력하면 go 버튼을 표시한다.

```
case State.TYPING_URL:
    btn_go.show ();
    break;
```

로딩 상태에서는 단순히 go 버튼을 다시 숨기고 refresh 버튼을 stop 버튼으로 변경한다.

```
case State.LOADING:
    btn_go.hide ();
    btn_refresh.label = "gtk-stop";
```

```
    break;
```

진행할 것인지 여부에 따라 forward 버튼을 활성화/비활성화로 설정한다.

```
if (view.can_go_forward ()) {
    btn_forward.sensitive = true;
} else {
    btn_forward.sensitive = false;
}
```

이와 비슷하게 진행 여부에 따라 backward 버튼을 활성화/비활성화로 설정한다.

```
if (view.can_go_back ()) {
    btn_back.sensitive = true;
} else {
    btn_back.sensitive = false;
}
```

이제 구성요소의 전체적인 상호작용을 준비하게 될 생성자를 살펴보자.

```
public WebBrowser ()
```

여기서 텍스트만 사용하는 대신 버튼에 대한 이미지를 사용한다.

```
Gtk.Settings.get_default ().gtk_button_images = true;
```

아래 코드에서는 단순히 .ui 파일을 열어 builder 객체로 삽입한다.

```
builder = new Builder ();
builder.add_from_file (UI_FILE);
```

아래 코드 조각에서는 웹 뷰를 인스턴스화하고 **Welcome** 텍스트를 표시한다. 텍스트는 gettext의 밑줄 함수로 닫아서(enclose) 번역 가능하게 만든다. HTML 헤딩 마크업의 번역은 무관하기 때문에 건너뛴다. 우리 관심사는 포맷팅 자체보다는 실제로 표시되는 텍스트에 있다.

```
view = new WebView();
view.load_string("<h1>" + _("Welcome") + "</h1>", "text/html", "UTF-8", "/");
```

다음으로 뷰의 load_started 시그널을 연결한다. 이는 링크를 클릭 시 호출되는 시그널로, 링크 연결 상태로 매핑된다. 하지만 이 시그널이 호출되고 나면 상태는 곧바로 로딩 상태로 이동한다. 이후 update_state 함수를 호출함으로써 UI의 모양을 업데이트한다.

```
view.load_started.connect(() => {
    state = State.LOADING;
    update_state();
});
```

하지만 이 함수는 Internet에서 이미지, 스크립트, 또는 다른 파일을 포함해 다른 소스를 로딩할 때마다 호출된다는 사실을 주목해야 한다.

load_finished 시그널은 로딩이 완료될 때마다 호출된다. 이런 경우 유휴 상태로 돌아간다.

```
view.load_finished.connect(() => {
    state = State.IDLE;
    update_state();
});
```

여기서는 URL 엔트리에서 키 누름 이벤트를 처리한다. 앞서 디자인한 바와 같이 무언가를 입력할 때마다 url 입력(typing url) 상태로 업데이트할 것이다. 따라서 이를 실행할 장소는 아래와 같다.

```
url_entry = builder.get_object ("url_entry") as Entry;
url_entry.key_press_event.connect(() => {
    state = State.TYPING_URL;
    update_state();
    return false;
});
```

아래 코드는 .ui 파일에 준비한 컨테이너로 뷰를 삽입하는 데에 그친다.

```
var box = builder.get_object ("webhost") as Container;
box.add(view);
```

다음으로 backward 버튼의 클릭 시그널을 처리한다. 버튼이 클릭되면 뷰에게 한 페이지씩 뒤로 갈 것을 요청한다.

```
btn_back = builder.get_object ("btn_back") as Button;
btn_back.clicked.connect(() => {
    view.go_back ();
});
```

Forward 버튼도 마찬가지다.

```
btn_forward = builder.get_object ("btn_forward") as Button;
btn_forward.clicked.connect(() => {
    view.go_forward ();
});
```

Refresh 버튼에는 이중 기능이 있으므로 먼저 새로고침 역할을 하는지, 아니면 중지 역할을 하는지 확인할 필요가 있다. 새로고침 역할을 한다면 뷰에서 reload를 호출하고, 중지 역할을 한다면 stop_loading을 호출한다.

```
btn_refresh = builder.get_object ("btn_refresh") as Button;
btn_refresh.clicked.connect(() => {
    if (btn_refresh.label == "gtk-refresh") {
        view.reload ();
    } else {
        view.stop_loading ();
    }
});
```

Go 버튼은 클릭 또는 활성화되어야 한다(Enter 누름). 두 액션 모두 URL 엔트리에 입력된 주소가 로딩되는 결과를 야기한다.

```
btn_go = builder.get_object ("btn_go") as Button;
btn_go.activate.connect(() => {
    view.load_uri (url_entry.text);
});

btn_go.clicked.connect(() => {
    view.load_uri (url_entry.text);
});
```

이는 단순히 창을 표시한다.

```
var window = builder.get_object ("window") as Window;
window.show_all ();
```

그리고 언제나처럼 창이 닫힐 때마다 애플리케이션을 종료할 것이다.

```
window.destroy.connect(() => {
    Gtk.main_quit();
});
```

상태를 초기화하기 위해 update_state를 호출할 것이다.

```
update_state();
```

꽤 간단한 코드다. 그렇지 않은가? 다음으로 할 일은 단위 테스트에서 update_state 함수를 테스트하고, 뷰의 스텁을 준비시키는 일이다. 테스트를 먼저 살펴보자. 여기서 모든 테스트는 TestWebBrowser 클래스에서 선언한다.

```
public class TestWebBrowser {
```

첫 번째 함수는 process_events 함수다. 이 함수는 제 12장, 품질이 좋으면 모든 게 쉬워진다에서 살펴본 함수와 정확히 동일하다. 큐에 대기 중인 이벤트를 모두 처리하고, 처리가 끝날 때까지 명시적으로 기다린다.

```
static void process_events()
{
    while (Gtk.events_pending ()) {
        Gtk.main_iteration_do(true);
    }
}
```

다음으로 이 테스트 함수에서 브라우저의 초기 상태를 테스트한다. WebBrowser 객체를 생성하고, 상태가 실제로 유휴 상태인지 확인한다. 그 다음 url 입력 상태를 테스트한다.

```
static void test_initial_state ()
{
    var web = new WebBrowser();
    process_events();
    assert (web.state == WebBrowser.State.IDLE);
}
```

```
```

이 함수에서는 WebBrowser 객체를 생성하고, show_all을 이용해 URL 엔트리와 창을 즉시 표시한다. 이는 위젯을 곧바로 사용할 수 있도록 확보하기 위함이다.

```
static void test_typing_url ()
{
    var web = new WebBrowser();
    var window = web.builder.get_object ("window") as Window;
    window.show_now ();
    web.url_entry.show_now ();
```

다음으로 go 버튼이 표시되는지 여부를 확인한다. 우리가 디자인한 바와 같이 숨겨져야 한다.

```
assert (web.btn_go.visible == false);
```

여기서는 마우스 포커스를 URL 엔트리로 두어 타이핑이 가능하도록 한다.

```
var entry_w = web.url_entry.get_window ();
web.url_entry.focus(0);
```

뭔가를 확인하기 전에는 유휴 상태에 있도록 해야 한다.

```
assert (web.state == WebBrowser.State.IDLE);
```

그리고 키 누름(과 해제를)을 시뮬레이트한다.

```
Gdk.test_simulate_key (entry_w, 5, 5, Gdk.Key.a, 0,
Gdk.EventType.KEY_PRESS);
Gdk.test_simulate_key (entry_w, 5, 5, Gdk.Key.a, 0,
Gdk.EventType.KEY_RELEASE);
```

이벤트가 실제로 위젯으로 전달되고 핸들러를 적절하게 호출되도록 하려면 초기에 호출해야 한다는 사실을 주목한다.

```
process_events ();
```

여기서는 url 입력 상태에 있는지 확인한다.

```
assert (web.state == WebBrowser.State.TYPING_URL);
```

이제 go 버튼이 표시되었는지 확인한다.

```
assert (web.btn_go.visible == true);
```

이후, 입력을 마치고 나면 포커스를 go 버튼으로 이동한다.

```
var btn_w = web.btn_go.get_window ();
web.btn_go.focus(0);
```

그리고 버튼에서 Enter 키를 누른다.

```
Gdk.test_simulate_key (btn_w, 5, 5, Gdk.Key.Return, 0,
Gdk.EventType.KEY_PRESS);
Gdk.test_simulate_key (btn_w, 5, 5, Gdk.Key.Return, 0,
Gdk.EventType.KEY_RELEASE);

process_events ();
```

웹 뷰가 페이지를 로딩하므로 상태는 로딩 상태로 변경되어야 한다.

```
assert (web.state == WebBrowser.State.LOADING);
```

두 개의 테스트를 갖게 되었다. 이제 main 함수를 살펴볼 차례다. 여기서 테스트 도구(test suite) 프레임워크를 초기화한다.

```
static int main (string[] args)
{
    Gtk.test_init (ref args);
```

앞서 정의한 함수로 매핑하는 두 개의 테스트를 등록한다.

```
Test.add_func ("/test-initial-state", test_initial_state);
Test.add_func ("/test-typing-url", test_typing_url);
```

이후 아이들 핸들러(idle handler)를 구성하여 Gtk.main을 호출한 직후 테스트가 실행될 수 있도록 한다. 모든 테스트가 실행되고 나면 애플리케이션을 즉시 중단한다.

```
Idle.add (() => {
    Test.run ();
    Gtk.main_quit ();
    return true;
});
```

여기서 이벤트 루프를 시작하고 GTK로 전달한다.

```
Gtk.main ();
```

테스트를 실행하기 위해 네트워크 연결을 사용하지 않음을 주목하라. 앞서 논했듯이 WebKit을 사용조차 하지 않는다. 대신 테스트 도중에 WebKit을 대신할 고유의 스텁을 사용한다.

우리 스텁에서는 스터빙하고자 하는 객체와 정확히 동일한 이름을 사용해야 한다. 또 WebKit 네임스페이스를 선언하여 using WebKit 절을 코드에서 사용할 수 있어야 한다. webkit_ 접두사를 이용해 C 코드를 생성할 것을 Vala에게 요청한다. 이러한 작업을 하지 않으면 Vala는 web_kit_ 접두사를 생성할 것이다(단어 사이에 밑줄을 주목).

```
[CCode (lower_case_cprefix = "webkit_")]
namespace WebKit {
```

이는 load_started와 load_finished 시그널에서 사용되는 헬퍼(helper) 클래스다.

```
[CCode (cheader_filename="webkit/webkit.h")]
public class WebFrame : Object {
}
```

그리고 변수와 시그널을 선언한다.

```
[CCode (cheader_filename="webkit/webkit.h")]
public class WebView : Viewport {
    public bool _can_go_back;
    public bool _can_go_forward;
    public signal void load_started (WebFrame frame);
    public signal void load_finished (WebFrame frame);
    WebFrame frame;
```

여기서는 단순히 frame 객체를 초기화한다.

```
public WebView() {
    frame = new WebFrame ();
}
```

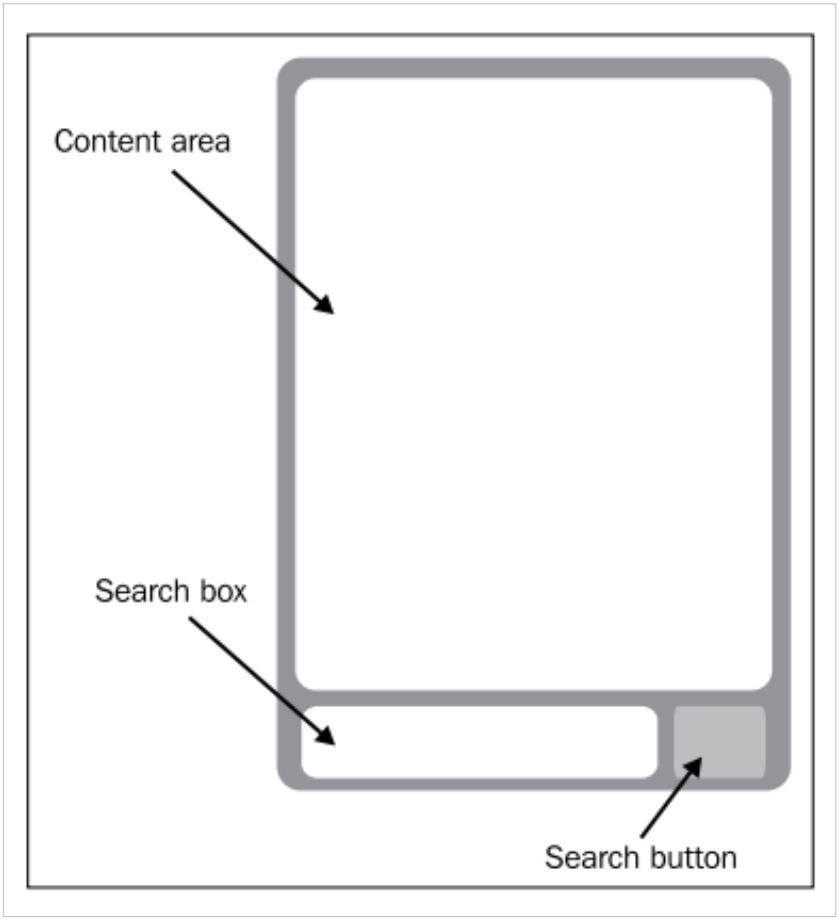
나머지는 꽤 간단할 뿐더러 제 12장, 품질이 좋으면 모든 게 쉬워진다편에서 이미 논했기 때문에 자세히 논하지 않겠다. 이 함수들은 빈 함수들로, 스텁에서는 구현 코드에서 실제로 사용되는 함수들만 정의한다.

시도해보기 - 더 많은 테스트 생성하기

이미 눈치 챘겠지만 우리에겐 두 가지 기능밖에 없다. 따라서 가능한 상태를 모두 테스트하지 못했다. 이제 가서 테스트를 더 만들어보자!

제 2부 - Twitter 클라이언트

이제 다음 주제인 트위터 클라이언트로 넘어가보자. 애플리케이션에는 우리가 검색하는 텍스트와 함께 트위트(tweet)의 스트림을 표시한다는 특징이 있다. 이는 웹 브라우저보다 더 간단하므로 Seed를 이용해 개발하도록 하겠다.



다시 말하지만 UI는 복잡하지 않다. 기본적으로 화면에 두 가지 부분이 있는데, 내용 영역(Content area)과 검색 영역(Search area)이다. 검색 영역 또한 두 가지 부분, 검색 박스(Search box)와 검색 버튼(Search button)으로 나뉜다. 버튼을 클릭할 때마다 검색 텍스트와 함께 트위트의 스트림으로 내용 영역이 업데이트된다.

실행하기 - Twitter 클라이언트 구현하기

세 개의 Seed 스크립트가 생길 것인데 이들을 디렉터리에 넣을 필요가 있다. 애플리케이션을 실행하기 위해서는 인터넷 연결이 필요함을 잊지 말라.

1. tweet-feed.js라는 새로운 스크립트를 생성하고 아래의 코드로 채워라.

```
#!/usr/bin/env seed

Gtk = imports.gi.Gtk;
Gdk = imports.gi.Gdk;
GObject = imports.gi.GObject;

f = imports.feedEntry;
t = imports.twitter;

var twitter = new t.Twitter();
```

```
Gtk.init(Seed.argv);
var window = new Gtk.Window();
window.resize(400,400);
window.title = "Tweet Feed";
var box = new Gtk.Box({orientation: Gtk.Orientation.VERTICAL});
window.add(box);

var scroll = new Gtk.ScrolledWindow();
var viewport = new Gtk.Viewport();
box.pack_start(scroll, true, true);

var search_box = new Gtk.Box({orientation: Gtk.Orientation.HORIZONTAL});
var entry = new Gtk.Entry();
entry.placeholder_text = "Enter your search topic";

search_box.pack_start(entry, true, true);
var search_button = new Gtk.Button({label: "gtk-find"});
search_button.use_stock = true;
search_button.signal.clicked.connect(function() {
    twitter.search(entry.text);
});
search_box.pack_start(search_button, false, false);

box.pack_start(search_box, false, false);

var entries = new Gtk.Box({orientation: Gtk.Orientation.VERTICAL});
scroll.add(viewport);
viewport.add(entries);
window.show_all();

twitter.signal.connect("data-available", function(object) {
    entries.foreach(function(content) {
        entries.remove(content);
    });
    for (var i = 0; i < twitter.data.results.length; i ++) {
        var entry = new f.FeedEntry(twitter.data.results[i]);
        entries.pack_start(entry, false, false);
    }
});

Gtk.main();
```

2. feedEntry.js라는 새로운 파일을 추가하고 아래의 코드를 사용하라.

```
Gtk = imports.gi.Gtk;
Gdk = imports.gi.Gdk;
GObject = imports.gi.GObject;
```

```
FeedEntry = new GType({
    parent: Gtk.TextView.type,
    name: "FeedEntry",
    properties: [
        {
            name: 'from_user_name',
            type: GObject.TYPE_STRING,
            default_value: "",
            flags: (GObject.ParamFlags.CONSTRUCT
                | GObject.ParamFlags.READABLE
                | GObject.ParamFlags.WRITABLE),
        }
        , {
            name: 'from_user',
            type: GObject.TYPE_STRING,
            default_value: "",
            flags: (GObject.ParamFlags.CONSTRUCT
                | GObject.ParamFlags.READABLE
                | GObject.ParamFlags.WRITABLE),
        }
        , {
            name: 'text',
            type: GObject.TYPE_STRING,
            default_value: "",
            flags: (GObject.ParamFlags.CONSTRUCT
                | GObject.ParamFlags.READABLE
                | GObject.ParamFlags.WRITABLE),
        }
    ],

    class_init: function(klass, prototype) {
        prototype.update_data = function() {
            var writer_tag = new Gtk.TextTag({
                name: "writer",
                size_points: 12,
                weight: 700,
                style: 3,
                foreground:"#000000"});
            this.buffer.tag_table.add(writer_tag);

            var user_tag = new Gtk.TextTag({
                name: „user",
                style: 0,
                foreground:"#888888"});
            this.buffer.tag_table.add(user_tag);

            var content_tag = new Gtk.TextTag({
```

```
                name: „content",
                size_points: 12,
                style: 0,
                weight: 400,
                foreground: „#000000"
                });
            this.buffer.tag_table.add(content_tag);

            this.buffer.insert_at_cursor (this.text + „\n", -1);
            var start_iter = this.buffer.get_start_iter ();
            var end_iter = this.buffer.get_end_iter ();
            this.buffer.apply_tag_by_name („content", start_iter.iter,
end_iter.iter);

            var cursor_pos = this.buffer.cursor_position;
            this.buffer.insert_at_cursor (this.from_user_name + " ",
-1);
            var start_iter =
this.buffer.get_iter_at_offset(cursor_pos);
            var end_iter = this.buffer.get_end_iter ();
            this.buffer.apply_tag_by_name ("writer", start_iter.iter,
end_iter.iter);
            var cursor_pos = this.buffer.cursor_position;
            this.buffer.insert_at_cursor ("@" + this.from_user + "\n",
-1);
            var start_iter =
this.buffer.get_iter_at_offset(cursor_pos);
            var end_iter = this.buffer.get_end_iter ();
            this.buffer.apply_tag_by_name ("user", start_iter.iter,
end_ iter.iter);
        }
    },
    init: function(self) {
        self.wrap_mode = Gtk.WrapMode.WORD;
        self.editable = false;
        self.update_data();
        self.show_all();
    }
});
```

3. twitter.js라는 새로운 파일을 또 추가하고 아래 코드로 채워라.

```
Gio = imports.gi.Gio;
GObject = imports.gi.GObject;

Twitter = new GType({
    parent: GObject.Object.type,
    name: "Twitter",
```

```
    signals: [
        {
            name: "data-available",
            parameters: []
        }
    ],
    class_init: function(klass, prototype) {
        prototype.search = function(keyword) {
            var url = "http://search.twitter.com/search.json?q=" +
keyword;
            var data_source = Gio.file_new_for_uri(url);
            var self = this;
            data_source.read_async(0, null,
                function(source, result) {
                    var input = source.read_finish(result);
                    var stream = new Gio.DataInputStream.c_new(input);
                    self.data = JSON.parse(stream.read_until("", 0));
                    self.signal["data-available"].emit();
                }
            );
        }
    },

    init: function(self) {
        this.data = {};
    }
});
```

4. 셸에 아래와 같은 명령을 입력함으로써 tweet-feed.js 파일에 실행 가능 권한을 부여하라.

```
chmod +x tweet-feed.js
```

5. 이를 실행하고 아무 검색 텍스트든 입력한 후 **Find** 버튼을 눌러라.

무슨 일이 일어났는가**?**

본문에서는 기존의 위젯을 확장하는 기술 뿐 아니라 네트워크로부터 자원을 읽는 지식도 이용하였다. 뿐만 아니라 개발을 모듈화하고 간소화하기 위해 다중 스크립트를 이용해 Seed 애플리케이션을 개발하는 방법도 학습하였다.

애플리케이션은 세 개의 모듈로 나누었다. 첫 번째는 메인 애플리케이션용이고, 두 번째는 트위터 데이터의 읽기용이며, 마지막은 트윗을 표시하기 위한 새로 파생된 위젯용이다.

마지막 위젯, 즉 FeedEntry 위젯부터 시작해보자. 먼저 TextView 위젯으로부터 새 위젯을 파생한다.

```
FeedEntry = new GType({
    parent: Gtk.TextView.type,
    name: "FeedEntry",
```

이를 파생하는 이유는 TextView에서 이용 가능한 텍스트 래핑 기능을 원하기 때문이다. 여기서 Twitter API로부터 비롯된 데이터를 정확하게 따르는 이름으로 몇 가지 프로퍼티를 정의한다.

```
properties: [
    {
        name: 'from_user_name',
        type: GObject.TYPE_STRING,
        default_value: "",
        flags: (GObject.ParamFlags.CONSTRUCT
            | GObject.ParamFlags.READABLE
            | GObject.ParamFlags.WRITABLE),
```

```
    }
```

API에는 "tweep"(트윕; 트윗을 게시한 사람)의 성명, 사용자명, 트윗 자체를 각각 포함하는 from_user_name, from_user, text가 있다. 따라서 Twitter API와 프로퍼티 간 직접적 맵을 새로운 위젯에서 이용하면 된다.

다음으로 프로토타입에 update_data 함수를 정의한다. 이는 함수를 클래스에서 이용할 수 있다는 뜻이다.

```
class_init: function(klass, prototype) {
    prototype.update_data = function() {
```

그리고 몇 가지 태그를 정의한다. 태그는 텍스트 버퍼에 입력하는 텍스트를 디자인하는 데에 사용된다.

```
var writer_tag = new Gtk.TextTag({
    name: "writer",
    size_points: 12,
    weight: 700,
    style: 3,
    foreground:"#000000"});
this.buffer.tag_table.add(writer_tag);
```

트윗에는 일반 스타일, 트윕의 이름에는 볼드체 스타일, 트위터 사용자 ID에는 연한(lighter) 글씨체를 사용하고자 한다. 하지만 태그를 사용하기 전에 먼저 태그를 태그 테이블로 추가한 후 모든 태그를 대상으로 실행한다. 이후 트윗을 먼저 삽입한 다음 새 행을 추가한다.

```
this.buffer.insert_at_cursor (this.text + "\n", -1);
```

여기서 Iter 객체를 얻음으로써 텍스트의 시작 위치를 기억한다. Iter 객체는 텍스트 버퍼 내 특정 위치를 가리키는 데 사용된다. 이것은 텍스트 선택내용(selection)에 비유하여 생각할 수 있겠다.

```
var start_iter = this.buffer.get_start_iter ();
```

그리고 이곳에서 텍스트 선택내용의 끝을 표시한다.

```
var end_iter = this.buffer.get_end_iter ();
```

이후 명시된 이름으로 된 태그를 시작과 끝 iters로 덮인 텍스트에 적용한다.

```
this.buffer.apply_tag_by_name ("content", start_iter.iter,
end_iter.iter);
```

선택내용의 비유로 돌아가면 이 부분은 텍스트 문서에서 선택된 텍스트로 포맷팅 스타일을 제공하는 것으로 생각할 수 있겠다.

여기서는 Iter 객체의 시작을 얻기 위해 더 이상 get_start_iter 함수를 사용할 수 없다.

```
var cursor_pos = this.buffer.cursor_position;
```

대신 현재 커서 위치를 얻어 아래에 삽입될 텍스트의 시작으로 표시한다.

다음으로 앞에서 기록한 커서 위치를 가리킴으로써 start iter를 얻는다.

```
this.buffer.insert_at_cursor (this.from_user_name + " ", -1);
var start_iter = this.buffer.get_iter_at_offset(cursor_pos);
```

이는 앞의 것과 동일하므로 그냥 텍스트의 끝을 이용해 end iter를 얻으면 된다.

```
var end_iter = this.buffer.get_end_iter ();
```

다음으로 iter 매개변수들 사이의 텍스트로 태그를 적용한다.

```
this.buffer.apply_tag_by_name ("writer", start_iter.iter,
end_iter.iter);
```

자연적으로 TextView는 텍스트 편집용이지만 여기서는 객체의 인스턴스 중 편집을 비활성화한다.

```
init: function(self) {
    self.wrap_mode = Gtk.WrapMode.WORD;
    self.editable = false;
```

초기화 중에 즉시 데이터를 업데이트하면 데이터가 렌더링된다.

```
self.update_data();
self.show_all();
```

이게 전부다. 이제 데이터 소스의 기능을 하는 twitter.js 파일을 살펴보자.

여기서 Twitter 객체를 간단한 GObject 객체로 정의한다.

```
Twitter = new GType({
    parent: GObject.Object.type,
    name: "Twitter",
```

data-available이라는 시그널이 있으며, 이는 어떠한 매개변수도 없이 호출된다.

```
signals: [
    {
        name: "data-available",
        parameters: []
    }
],
```

시그널은 우리가 Twitter API로부터 JSON 포맷의 새로운 데이터가 들어올 때마다 호출된다. 이는 클래스에서 이용 가능하게 만드는 search 함수다. 이 함수는 언급된 URL로부터 스트림을 얻어 뒤에 키워드를 추가하는 일을 한다. 스트림은 Gio를 이용해 연다.

```
class_init: function(klass, prototype) {
    prototype.search = function(keyword) {
        var url = "http://search.twitter.com/search.json?q=" + keyword;
```

```
        var data_source = Gio.file_new_for_uri(url);
        var self = this;
```

이후 비동기식으로 스트림을 읽어 인터넷으로부터 데이터를 로딩하는 동안 UI를 방해하지 않도록 한다. 데이터가 준비되면 read_async 함수 뒤에 클로져(closure)를 넣어 함수를 입력한다. 여기서 스트림은 DataInputStream 인스턴스로 변환되고, 우리는 데이터를 JavaScript 객체로 파싱한다. 모든 것이 준비되면 data-available 시그널을 발생시킨다.

```
data_source.read_async(0, null,
    function(source, result) {
        var input = source.read_finish(result);
        var stream = new Gio.DataInputStream.c_new(input);
        self.data = JSON.parse(stream.read_until("", 0));
        self.signal["data-available"].emit();
    }
);
```

우리 초기화 함수는 이토록 간단하다. 그저 data member를 빈 객체로 초기화할 뿐이다.

```
init: function(self) {
    this.data = {};
}
```

다음으로 tweet-feed.js에서 메인 애플리케이션 코드를 살펴보자. 여기서는 다른 두 개의 스크립트를 가져와 f와 t 변수로부터 접근 가능하게 만든다.

```
f = imports.feedEntry;
t = imports.twitter;
```

코드 베이스가 커지면 이는 바람직하지 못한 예제가 된다. feedEntryScript 또는 twitterScript와 같은 이름을 사용하는 편이 더 낫다.

아래 행에서는 Twitter 객체를 인스턴스화한다.

```
var twitter = new t.Twitter();
```

그리고 Gtk를 초기화하여 창을 생성한다. 처음 크기를 설정하고 제목을 제공한다.

```
Gtk.init(Seed.argv);
var window = new Gtk.Window();
window.resize(400,400);
window.title = "Tweet Feed";
```

레이아웃으로 사용될 Box 객체가 있으며, 이는 창으로 넣는다.

```
var box = new Gtk.Box({orientation: Gtk.Orientation.VERTICAL});
window.add(box);
```

이제 창에 하나 이상의 위젯을 추가할 수 있다.

여기에는 viewport와 함께 생성된 scroll이라 불리는 새로운 ScrolledWindow 객체가 있다. 창의 크기를 넘어갈 때 트윗을 스크롤하려면 이를 꼭 실행해야 한다.

```
var scroll = new Gtk.ScrolledWindow();
var viewport = new Gtk.Viewport();
box.pack_start(scroll, true, true);
```

이후 새로운 수평적 박스를 화면의 하단에 추가한다. 텍스트 엔트리와 검색 버튼의 위치일 것이다.

```
var search_box = new Gtk.Box({orientation: Gtk.Orientation.HORIZONTAL});
```

여기서는 사용자가 검색 텍스트를 입력하도록 텍스트 엔트리를 생성한다. 사용자가 해야 할 일을 말해주는 힌트로 플레이스 홀더 텍스트를 사용한다. 그리고 수평적 박스로 엔트리를 패킹한다.

```
var entry = new Gtk.Entry();
entry.placeholder_text = "Enter your search topic";
search_box.pack_start(entry, true, true);
```

여기서는 검색 버튼을 생성한다. 스톡 항목이 있는 간단한 버튼이므로 라벨을 생성할 필요가 없다. 다음으로 twitter.search 함수를 호출하고 스트림 다운로드를 시작함으로써 클릭된 시그널을 연결한다.

```
var search_button = new Gtk.Button({label: "gtk-find"});
search_button.use_stock = true;
search_button.signal.clicked.connect(function() {
    twitter.search(entry.text);
});
search_box.pack_start(search_button, false, false);
```

게다가 들어오는 트윗을 모두 관리하기 위한 수직 박스를 생성한다. 박스는 스크롤이 있는 창 안에 위치한 뷰포트에 놓인다.

```
var entries = new Gtk.Box({orientation: Gtk.Orientation.VERTICAL});
scroll.add(viewport);
viewport.add(entries);
```

이것은 data-available 시그널 핸들러다. 앞서 언급했듯이 시그널은 우리가 전체 트위터 스트림을 수신할 때 발생된다.

```
twitter.signal.connect("data-available", function(object) {
    entries.foreach(function(content) {
        entries.remove(content);
    });
    for (var i = 0; i < twitter.data.results.length; i ++) {
        var entry = new f.FeedEntry(twitter.data.results[i]);
        entries.pack_start(entry, false, false);
    }
});
```

따라서 우리는 기존의 엔트리가 있다면 이를 먼저 제거해야 한다. 그리고 twitter.data.results 배열에 있는 트윗을 모두 반복한다. 각 트윗에는 단순히 FeedEntry 객체를 생성한다. 엔트리 구조체는 직접 전달해야 하는데, 구조체의 member가 이미 FeedEntry 클래스에 매핑되어 있기 때문이다. 이후 entries 배열에 엔트리 객체를 추가한다.

요약

이제 두 개의 실제 애플리케이션이 있다. 하지만 여전히 이곳저곳에 놓친 부분들이 있다. 그리고 이 부분들은 당신이 수정해주길 기다리고 있다.

첫 번째 프로젝트에서는 configure 스크립트로부터 경로를 직접 하드코딩함으로써 지금 개발 모드인지 아닌지를 확인하는 코드의 생성을 피하는 새로운 트릭을 학습하였다. 또 웹 페이지를 로딩하는 방법과 크롬의 나머지 부분과 상호작용하는 방법도 논했다.

후에 두 번째 프로젝트에서는 위젯과 다중 스크립트 애플리케이션을 확장하는 방법을 학습하였다. 이는 알아두면 중요한 내용인 것이, 실제 세계에서 애플리케이션이 이 상황과 비슷할 수 있기 때문이다.

두 프로젝트에서 우리는 UI가 아닌(non-UI) 코드를 전용 클래스에 넣음으로써 논리와 UI 빌드를 분리하는 것을 논했다. 이를 통해 UI 코드로 혼합되었기 때문에 좀 더 복잡한 비지니스 프로세스를 쉽게 추가하면서도 혼동하지 않을 수 있다.

축하한다. 이제 책의 마지막에 도달했다! 이제 다음으로 할 일은 무엇일까?

책을 마쳤다 하더라도 지금까지 읽은 표면 지식의 세부내용을 파고 들어갈 필요가 있다. 하지만 필수 내용은 이미 이해했으니 세부내용을 학습하기가 수월할 것이다. 한 가지 기억해야 할 중요한 사항은 GNOME 플랫폼이 오픈 소스 프로젝트란 사실이다. 여느 다른 프로젝트와 마찬가지로 전 세계 해커들이 기여한 덕분이다. 이제 당신도 그 이상의 일을 할 수 있다!

Notes

GNOME3ApplicationDevelopmentBeginnersGuide:App

팝 퀴즈 정답

팝 퀴즈 정답

제 **2**장**,** 무기 준비하기

팝 퀴즈 **-** 시그널 명명하기

Q1

정답: server_on_connection_started. 첫 번째 단어는 클래스(서버)를 나타내고, 그 뒤의 on은 특정 유형의 이벤트가 발생하고 있음을 의미한다.

제 **3**장**,** 프로그래밍 언어

팝 퀴즈 **-** 이를 어떻게 고칠까**?**

Q1

정답: public. Book 클래스 외부에 있는 Main 클래스로부터 접근하길 원하기 때문이다.

팝 퀴즈 **-** 이제 값은 무엇인가**?**

Q1

정답: 정의되지 않은 값으로부터 .length로 접근을 시도하므로 JavaScript는 오류라고 생각할 것이다.

팝 퀴즈 **-** 차이점이 눈에 띄는가**?**

Q1

정답: Circle은 정의를 갖고 있기 때문에 클래스이고, circle은 Circle 클래스로부터 인스턴스화된 객체다.

팝 퀴즈 **-** 전역적으로 만드는 방법은**?**

Q1

정답: Book 클래스 프로토타입에서 printAuthor를 추가하면 Book으로부터 생성된 모든 객체는 함수를 갖게 될 것이다. 모든 인스턴스화된 객체에 메서드를 넣는다면 너무 번거로운 반면 프로토타입에 넣으면 메서드가 항상 생성되도록 보장할 수 있다.

제 **4**장**, GNOME** 코어 라이브러리 사용하기

팝 퀴즈 **-** 왜 **0** 값이 출력되는가

Q1

정답: 둘 다 맞는 답이다. 기본값은 생성 단계에서 설정된다.

제 **5**장**,** 그래픽 사용자 인터페이스 애플리케이션 빌드하기

Q1

정답: 약간 왼쪽.

0.3은 0.5보다 작기 때문이다. 0.0 값은 완전히 왼쪽으로 정렬됨을 의미한다.

Q2

제시된 답들 중 정답은 없으며, 약간 위쪽이 올바른 답이다.

0.3은 수직적으로 값이 0.0인 최상단에 더 가까이 위치함을 의미한다.

제 **10**장**,** 데스크톱 통합

팝 퀴즈 **-** 훌륭한 애플리케이션 예제

Q1

정답: 배터리 시스템 트레이 애플릿. 데이터는 항상 게시되기 때문에 시스템 트레이 애플릿과 같이 애플리케이션을 감시하는 데에 좋다. 배터리 확인 애플리케이션은 항상 폴링(poll)하는 것이 아니라 데이터를 한 번 또는 필요 시에만 한다.

Notes

문서 출처 및 기여자

GNOME3ApplicationDevelopmentBeginnersGuide:Copyright 출처: http://trans.onionmixer.net/mediawiki/index.php?oldid=5125 기여자: Onionmixer

GNOME3ApplicationDevelopmentBeginnersGuide:Credits 출처: http://trans.onionmixer.net/mediawiki/index.php?oldid=5126 기여자: Onionmixer

GNOME3ApplicationDevelopmentBeginnersGuide:AbouttheAuthor 출처: http://trans.onionmixer.net/mediawiki/index.php?oldid=5127 기여자: Onionmixer

GNOME3ApplicationDevelopmentBeginnersGuide:AbouttheReviewers 출처: http://trans.onionmixer.net/mediawiki/index.php?oldid=5128 기여자: Onionmixer

GNOME3ApplicationDevelopmentBeginnersGuide:PacktPub 출처: http://trans.onionmixer.net/mediawiki/index.php?oldid=5137 기여자: Onionmixer

GNOME3ApplicationDevelopmentBeginnersGuide:Preface 출처: http://trans.onionmixer.net/mediawiki/index.php?oldid=5136 기여자: Onionmixer

GNOME3ApplicationDevelopmentBeginnersGuide:Chapter 01 출처: http://trans.onionmixer.net/mediawiki/index.php?oldid=5146 기여자: Onionmixer

GNOME3ApplicationDevelopmentBeginnersGuide:Chapter 02 출처: http://trans.onionmixer.net/mediawiki/index.php?oldid=5163 기여자: Onionmixer

GNOME3ApplicationDevelopmentBeginnersGuide:Chapter 03 출처: http://trans.onionmixer.net/mediawiki/index.php?oldid=5201 기여자: Onionmixer

GNOME3ApplicationDevelopmentBeginnersGuide:Chapter 04 출처: http://trans.onionmixer.net/mediawiki/index.php?oldid=5202 기여자: Onionmixer

GNOME3ApplicationDevelopmentBeginnersGuide:Chapter 05 출처: http://trans.onionmixer.net/mediawiki/index.php?oldid=5208 기여자: Onionmixer

GNOME3ApplicationDevelopmentBeginnersGuide:Chapter 06 출처: http://trans.onionmixer.net/mediawiki/index.php?oldid=5213 기여자: Onionmixer

GNOME3ApplicationDevelopmentBeginnersGuide:Chapter 07 출처: http://trans.onionmixer.net/mediawiki/index.php?oldid=5220 기여자: Onionmixer

GNOME3ApplicationDevelopmentBeginnersGuide:Chapter 08 출처: http://trans.onionmixer.net/mediawiki/index.php?oldid=5228 기여자: Onionmixer

GNOME3ApplicationDevelopmentBeginnersGuide:Chapter 10 출처: http://trans.onionmixer.net/mediawiki/index.php?oldid=5250 기여자: Onionmixer

GNOME3ApplicationDevelopmentBeginnersGuide:Chapter 12 출처: http://trans.onionmixer.net/mediawiki/index.php?oldid=5251 기여자: Onionmixer

GNOME3ApplicationDevelopmentBeginnersGuide:Chapter 13 출처: http://trans.onionmixer.net/mediawiki/index.php?oldid=5260 기여자: Onionmixer

GNOME3ApplicationDevelopmentBeginnersGuide:Appendix 출처: http://trans.onionmixer.net/mediawiki/index.php?oldid=5262 기여자: Onionmixer

그림 출처, 라이선스와 기여자

image:GNOME01_preface_packt.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME01_preface_packt.png 라이선스: 알 수 없음 기여자: Onionmixer
image:gnome3_notice_header.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:Gnome3_notice_header.png 라이선스: 알 수 없음 기여자: Onionmixer
image:gnome3_notice_footer.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:Gnome3_notice_footer.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter01_01.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter01_01.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter01_02.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter01_02.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter01_03.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter01_03.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter01_04.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter01_04.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter01_05.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter01_05.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter01_06.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter01_06.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter01_07.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter01_07.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter02_01.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter02_01.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter02_02.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter02_02.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter02_03.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter02_03.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter02_04.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter02_04.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter02_05.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter02_05.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter02_06.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter02_06.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter02_07.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter02_07.png 라이선스: 알 수 없음 기여자: Onionmixer
image:gnome3_light_header.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:Gnome3_light_header.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter02_08.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter02_08.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter02_09.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter02_09.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter02_10.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter02_10.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter02_11.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter02_11.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter02_12.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter02_12.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter02_13.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter02_13.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter03_01.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter03_01.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter03_02.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter03_02.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter03_03.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter03_03.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter03_04.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter03_04.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter03_05.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter03_05.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter04_01.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter04_01.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter04_02.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter04_02.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter04_03.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter04_03.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter05_01.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter05_01.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter05_02.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter05_02.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter05_03.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter05_03.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter05_04.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter05_04.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter05_05.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter05_05.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter06_01.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter06_01.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter06_02.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter06_02.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter06_03.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter06_03.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter06_04.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter06_04.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter07_01.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter07_01.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter07_02.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter07_02.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter07_03.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter07_03.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter07_04.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter07_04.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter07_05.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter07_05.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter07_06.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter07_06.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter08_01.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter08_01.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter08_02.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter08_02.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter08_03.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter08_03.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter08_04.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter08_04.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter08_05.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter08_05.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter08_06.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter08_06.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter08_07.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter08_07.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter10_01.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter10_01.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter10_02.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter10_02.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter10_03.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter10_03.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter10_04.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter10_04.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter13_01.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter13_01.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter13_02.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter13_02.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter13_03.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter13_03.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter13_04.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter13_04.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter13_05.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter13_05.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter13_06.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter13_06.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter13_07.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter13_07.png 라이선스: 알 수 없음 기여자: Onionmixer
image:GNOME3_Chapter13_08.png 출처: http://trans.onionmixer.net/mediawiki/index.php?title=파일:GNOME3_Chapter13_08.png 라이선스: 알 수 없음 기여자: Onionmixer