8장

자바 빈스™

- 빈을 사용하는 이유는?
- 빈스 작성 절차
- BDK와 빈박스(BeanBox)
- 빈으로 이미지 뷰어 응용 프로그램 개발
- 빈 속성과 이벤트를 위한 명명 규칙
- 빈 속성 타입
- 속성 편집기
- 명명 규칙을 넘어-BeanInfo 클래스 생성하기
- 커스터마이져들
- 빈 컨텍스트

자바 빈 명세에서 제공한 빈의 공식적인 정의는 다음과 같다: "빈은 빌더 도구에서 가시적으로 조작할 수 있는 선의 자바 빈 명세에 기반한 재사용 가능한 소프트웨어 컴포넌트이다"

일단 여러분이 빈을 구현하면 다른 사람들은 Jbuilder나 VisualAge 또는 VisualCafe 같은 빌더 환경내에서 응용 프로그램이나 애플릿을 보다 효율적으로 개발하기 위해 사용할 수 있다.

우리는 이러한 환경들을 사용하는 방법에 대해서는 말하지 않을 것이다 – 여러분은 벤더들이 제공하는 문서들을 참조해야 한다. 이 장은 다른 프로그래머들이 쉽게 여러분의 빈을 사용할 수 있도록 빈을 구현하기 위해 알아야 할 것들에 대해 설명한다.

빈을 사용하는 이유는?

윈도우에 대한 배경지식을 갖고 있는 프로그래머들은, 특히 비쥬얼 베이직이나 델파이 같은, 왜 빈이 이렇게 중요한지 즉시 알아챌 것이다. 모든 것은 여러분 "자신의 역할"이다라는 전통적인 방식의 환경에 익숙한 프로그래머들은 즉시 알아채지는 못할 것이다. 우리의 경험으로 볼 때, 비쥬얼 베이직에 대한 배경지식이 없는 프로그래머들은 비쥬얼 베이직이 재사용 가능한 객체 기술의 가장 성공적인 예제들 중 하나라는 사실을 믿기 어렵다는 발견한다. 비쥬얼 베이직이 인기있는 이유중 하나는 비쥬얼 베이직 응용 프로그램을 개발하는 방법을 분명하게 제공한다는 것이다. 아직까지 비쥬얼 베이직으로 작업한 적이 없는 사람들을 위해 여기서 여러분이 무엇을 해야 할지 보여준다.

1. 폼 윈도우내에 (비쥬얼 베이직에서 *컨트롤*이라 불리는) 컴포넌트들을 떨어뜨리는 것에 의해서 인터페이스를 구성한다.
2. *속성 시트*를 통해 높이, 색상, 또는 다른 행위 같은 컴포넌트의 속성을 설정한다.
3. 또한 속성 시트는 컴포넌트들이 반응할 수 있는 이벤트들을 나열한다. 이러한 몇몇 이벤트들을 위해, 여러분은 이벤트 핸들링을 위한 작은 조각의 코드를 작성한다.

예를 들면, 제1권의 2장에서 우리는 프레임에서 이미지를 출력하는 프로그램을 작성했다. 이것은 한 페이지의 코드를 약간 넘는다. 다음은 같은 기능을 갖는 보다 예쁜 프로그램을 비쥬얼 베이직으로 작성할 때 여러분이 해야 할 것들이다.

1. 윈도우에 두 개의 컨트롤을 추가한다: 그래픽을 출력하기 위한 Image 컨트롤과 파일을 선택하기 위한 CommonDialog 컨트롤.
2. 오직 Image 컨트롤이 처리할 수 있는 파일들 만을 보여주도록 CommonDialog 컨트롤의 Filter 속성을 설정한다. VB에서 이 작업은 그림8-2에서 보여지는 것처럼 속성 윈도우를 호출하면 된다.

그림8-1 : 이미지 응용 프로그램을 위한 VB의 속성 윈도우

이제, 우리는 프로젝트가 첫번째로 수행되기 시작할 때 작동될 5줄의 VB 코드를 작성할 필요가 있다. Form_Load라 불리는 이벤트가 여기에 해당한다, 그래서 Form_Load내의 코드가 이벤트를 처리한다. 다음 코드는 파일 다이얼로그 박스를 팝업하는 코드이다 – 그러나 적당한 파일 확장자를 갖는 파일만이 보여진다. 왜냐하면 필터를 적당하게 설정했기 때문이다. 사용자가 이미지 파일을 설정한 다음에 코드는 이미지 컨트롤에게 이를 출력하라고 말한다. 이 과정을 위해 필요한 모든 코드는 다음가 같다:

```
Private Sub Form_Load()
        On Error Resume Next
        CommonDialog1.ShowOpen
        Image1.Picture = LoadPicture(CommonDialog1.FileName)
End Sub
```

이것이 전부이다. 레이아웃이 작동하고 다섯 줄의 코드가 결합되어 자바 프로그래밍 언어 코드의 페이지와 같은 기능을 제공한다. 분명, 컴포넌트를 가져다 떨어뜨리는 방법과 코드를 작성하는 것보다 속성을 설정하는 것을 배우는 게 한결 쉽다.

현재 시점에서 중요한 것은 자바 프로그래밍 언어가 여전히 최상의 객체지향 프로그래머들

에게 가장 각광받는 도구라는 사실이다. 그리고 실제 이러한 프로그래머들은 중.소규모의 응용 프로그램 사용자 인터페이스를 디자인 하기위해 비쥬얼 베이직을 사용하는 프로그래머만큼의 생산성을 좋아하지 않는다. (주: 실제 고급 프로그래머들은 자동화된 도구를 통해 지저분한 코드가 생성되는 걸 좋아하지 않기 때문에 직접 코딩하길 더 선호한다.) 자바 빈 기술은 벤더들로 하여금 자바 응용 프로그램을 위한 사용자 인터페이스를 개발하기 쉽게 만들어 준다.

우리는 비쥬얼 베이직이 모든 문제를 해결해 주는 좋은 도구라는 것을 강조하길 원하지 않는다. 비쥬얼 베이직은 특정한 종류의 문제들을 해결하기 위해 최적화되었다-UI기반의 윈도우 프로그래밍. 상대적으로 자바 기술은 항상 보다 넓은 범주의 용도로 사용된다. 자바 빈 기술의 출현은 좋은 뉴스이다, 비쥬얼 베이직 스타일의 응용 프로그램을 작성하는 것의 몇 가지 이점이 프로그래머들에게 공평하게 적용된다. 특히:

1. 가장 일반적인 비쥬얼 베이직 컨트롤과 같은 기능을 갖는 빈들을 지속적으로 이용할 수 있다.
2. 자바 빌더 도구들은 비쥬얼 베이직 환경을 사용하는 것만큼 쉽다.

노트 : 일단 여러분이 빈을 작성하면, 자바 빈 기술을 지원하는 모든 환경의 사용자들은 실제 이 빈을 이용할 수 있다. 심지어 빈들은 "빈에서 액티브X로의 변환 브리지"의 기술을 통해 비쥬얼 베이직과 델파이내에서 사용할 수 있다(http://java.sun.com/beans/bridge).

빈스 작성 절차

이 장의 나머지 대부분은 빈을 작성하는 데 사용할 기술들을 보여준다. 보다 자세히 살펴보기 전에 우리는 전반적인 과정을 먼저 제공한다. 먼저 우리는 빈을 작성하는 것이 기술적인 어려움이 아니라는 것을 강조하고 싶다.- 여러분이 마스터해야 할 새로운 클래스와 인터페이스는 아주 적은 수이다.

특히, 가장 단순한 종류의 빈은 실제 이벤트 리스너들과 메소드들을 위한 몇가지 아주 엄격한 명명 규칙을 준수하는 자바 플랫폼 클래스에 지나지 않는다. 예제 8-1은 앞선 절에서 소개한 비쥬얼 베이직 이미지 컨트롤과 같은 기능의 자바 기술에 기반한 빌더 환경에서 제공하는 ImageViewer 빈을 위한 코드이다.

예제 8-1 : ImageViewerBean.java

```
import java.awt.*;
```

```
import java.io.*;
import javax.swing.*;

public class ImageViewerBean extends JPanel
    implements Serializable
{   public void setFileName(String f)
    {   fileName = f;
        image = Toolkit.getDefaultToolkit().getImage(fileName);
        MediaTracker tracker = new MediaTracker(this);
         tracker.addImage(image, 0);
        try { tracker.waitForID(0); }
        catch (InterruptedException e) {}
         repaint();
    }

    public String getFileName()
    {   return fileName;
    }

    public void paint(Graphics g)
    {   if (image == null)
        {   g.drawRect(0, 0, getWidth() - 1, getHeight() - 1);
        }
        else
           g.drawImage(image, 0, 0, this);
    }

    public Dimension getPreferredSize()
    {   if (image == null)
            return new Dimension(MINSIZE, MINSIZE);
        return new Dimension(image.getWidth(null),
             image.getHeight(null));
    }

   private static final int MINSIZE = 50;
   private Image image = null;
```

```
  private String fileName = "";
}
```

여러분이 이 코드를 볼 때 거의 잘 디자인된 자바 플랫폼 클래스와 크게 다르지 않다는 것을 알게 될 것이다. 예를 들면, 모든 접근 메소드들은 get으로 시작하고 모든 변경 메소드들은 set으로 시작한다. 여러분이 보게 될 것처럼 빌더 도구는 프로퍼티를 찾기 위해 이러한 표준 명명 규칙을 사용한다. fileName은 앞선 예제 빈의 속성이다. 왜냐하면 get과 set 메소드를 갖고 있기 때문이다. 이 특별한 예제에서 fileName 속성은 fileName 이라 불리는 인스턴스 변수내에 저장된다. 그러나 속성은 인스턴스 변수내에 저장될 필요가 없다-get 메소드는 다른 몇가지 방법으로 이를 계산할 수 있다.

노트 : 빈 환경을 완전히 지원하기 위해, 빈들은 기본 생성자(또는 자동적으로 제공되는 기본 생성자를 이용하기 위해 생성자를 제공하지 않아야 함)를 가져야 하고 연속화 돼야 한다. 이들은 환경이 새로운 빈들을 초기화하고 세션들 사이의 빈들을 저장하게 한다. 그러나 이것은 추천사항이지 요구사항은 아니다.

여러분들이 이 장의 예제들을 읽을 때 한가지 명심해야 할 점은 실세계에 적용하는 빈들은 다음의 두 이유에서 우리의 간단한 예제보다 훨씬 복잡하고 장황하다는 것이다.

1. 빈들은 덜 숙련된 프로그래머들에 의해서 사용되어질 필요가 있다. 이 같은 사람들은 프로그래밍을 반드시 해야 할 필요가 없는 비쥬얼 디자인 도구로 여러분이 제공하는 빈의 모든 기능에 접근할 것이다. 여러분은 빈의 행동을 쉽게 커스터마이징하기 위해 많은 속성을 노출시킬 필요가 있다.
2. 같은 빈이 다양한 분야나 내용으로 널리 사용될 필요가 있다. 실세계에 사용하기에 너무 단순한 빈은 인정받지 못한다. 물론, 이러한 이유에서 많은 사람들이 챠트 컨트롤 같은 전문 컴포넌트를 이용하기 위해 수백 달러를 지불한다. 예를 들면, 이 글을 쓰고 있는 시점에 가장 인기높은 챠트 컨트롤은 다음을 가지고 있다:

    - 60개의 속성
    - 47개의 이벤트
    - 14의 메소드
    - 178 페이지의 문서

분명, 챠트 컨트롤은 상상할 수 있는 모든 기능의 컨트롤을 이다. 그러나 비록 숫자들을 입력하는 텍스트 필드처럼 간단한 빈일지라도 최초로 상상한 것 같은 단순한 코드는 아니라는

것이 판명되었다.

가령, 텍스트 필드에 사용자가 숫자를 입력한다고 가정해 보자. 새로운 숫자들이 즉시 리스너에게 보고 될까? 다른 한편으로 이러한 접근 방법은 빈의 사용자에게 즉각적인 피드백을 제공하기 때문에 아주 매력적이다. 그러나 많은 응용 프로그램은 유효 항목들에 대한 범위를 제약하고 최종 유효 항목에 포함되는 일부 항목들은 그 자체가 유효하지 않을 수 있다.

가령, 리스너가 입력 필드로부터 오직 짝수만을 받길 원한다고 가정해 보자. 만일, 사용자가 256을 입력하려 한다면 리스너는 유효하지 않은 입력 25를 거부하고 사용자는 결코 통과하지 못한다. 그래서 이 경우, 점진적인 갱신은 나쁜 아이디어이다. 왜냐하면 증가하는 값은 최종적인 값이 유효하더라도 거부되기 때문이다. 이러한 경우, 여러분의 빈은 사용자가 ENTER 키를 누르거나 텍스트 필드가 포커스를 잃어버렸을 때 처럼 사용자가 숫자 입력을 마쳤을 때 오직 이 값을 리스너에게 보고해야 한다.

만일 여러분이 단일 응용 프로그램을 위한 프로그램을 작성한다면 하나 또는 다른 행동을 선택할 것이다. 그러나 만일 여러분이 빈을 작성한다면 여러분 자신이 아니라 다른 요구사항을 갖는 넓은 범위의 사용자를 위한 프로그래밍이어야 한다. 그러므로 incrementalUpdate같은 프로퍼티를 추가하고 이 프로퍼티의 값에 따른 행위를 수행하도록 구현하는 것이 최선의 방법이다. 이러한 옵션들을 추가하고 나면 단순한 IntTextBean조차도 12개의 특성을 갖게 된다.

운좋게도, 여러분은 작은 수의 개념만을 마스터하면 많은 행위를 갖는 빈들을 작성할 수 있다. 이 장의 예제 빈들은 단순하지만 필요한 개념들을 예증하기 위해 충분하다.

팁 : 여러분은 많은 프로그래밍 제품을 취급하는 상점과 자바 개발 환경에서 제공하는 번들, 또는 무료로 제공하는 www.gamelan.com이나 IBM 알파웍스(http://alphaworks.ibm.com)에서 전체 기능을 제공하는 빈을 발견할 수 있다. 모든 스윙 컴포넌트들 또한 빈이다.

BDK와 빈박스(BeanBox)

빈을 작성하는 방법으로 바로 들어가기 전에 , 우리는 여러분들이 빈을 사용하고 이들을 테스트 하는 방법을 익히길 원한다. ImageViewerBean은 완전하게 사용가능한 빈이다. 그러나 빌더 환경 외에서는 특별한 기능들이 보여지지 않는다. 특히, 자바 프로그래밍 언어로 작성된 일반 프로그램에서 이를 사용하는 유일한 방법은 이를 초기화하는 클래스를 작성하고

setFileName 메소드를 호출하는 것이다. 그래서 질문은 어떻게 이러한 단순한 빈의 기능을 보여줄 것인가? 라는 것이다. 반대로 이 질문은 어느 빌더 도구를 사용할 것인가? 라는 질문을 이끌어 낸다. 수많은 좋은 환경들이 이용가능하다. 그러나 이들은 서로 다르게 작동한다. 특정 벤더 하나의 제품을 선호하는 것보다 우리는 빈 박스(BeanBox)라 불리는 공통의 환경을 사용한다. 빈 박스는 빈 개발 킷(BDK)의 일부이다. 빈 박스는 모든 기능을 제공하지는 않지만 많은 플랫폼상에서 이용가능하고 단순한 빈을 검사할 수 있게 해준다. 더욱이 여러분이 빈 박스내에서 작동하는 빈을 얻는 다면 어떤 변경도 필요없이 다른 개발 환경내에서 작동될 것이다.

노트 : 현재 BDK 설치는 JDK설치와는 별도로 설치해야 한다. 여러분은 자바 웹 사이트 (http://java.sun.com/beans)에서 BDK를 다운로드 받을 필요가 있다. 이 장을 계속 진행하기 전에 미리 BDK를 설치해야 한다는 점을 기억하라.

경고 : 윈도우상에서 BDK를 Program Files 디렉토리하에 설치해서는 안된다. 만일 설치한다면 디렉토리 이름내의 공백이 여러분을 우울하게 만들 것이다. BDK를 bdk나 bdk1.1같은 이름의 디렉토리내에 설치하라.

빈 박스 사용하기

우리는 여러분이 bdk 디렉토리에 BDK를 설치했다고 가정한다. 빈박스는 자동적으로 beanbox라 불리는 하위 디렉토리내에 설치된다. 이 디렉토리는 윈도우하에서 빈 박스를 구동하기 위한 배치 파일(run.bat)과 유닉스를 위한 쉘 스크립트(run.sh)을 가지고 있다. 빈 박스를 구동할 때 여러분은 그림8-2같은 메시지 박스를 보게 될 것이다. 이 메시지는 빈 박스가 JAR 파일들을 로드하는 것을 나타낸다- 여러분이 보게 될 것처럼 빈들은 JAR 파일들내에 묶여진다.(JAR 파일들에 대한 자세한 정보는 1권의 10장을 참조하라.)

그림 8-2: “Analyzing Jars” 메시지 박스

실제 빈 박스는 그림8-3에서 보여지는 것처럼 네 개의 독립적인 크기 조절가능하며 이동 가능한 윈도우로 구성된다. 툴 박스(ToolBox)는 빈 박스가 알고 있는 모든 빈들이 목록을 나열한다.(새로운 빈을 빈 박스에 추가하는 방법은 아래를 참조하라.) 그림8-3에서 볼 수 있듯이 몇몇 빈들은 툴 박스의 왼편에 보여지는 연관된 아이콘들을 가지고 있다. 빈 박스라고 표기

된 윈도우는 빈들을 떨어뜨리고 이들 간의 하나의 트리거 액션내의 이벤트들을 연결할 공간이다. 프로퍼티 빈 박스라고 표기된 윈도우는 여러분의 빈들의 속성을 조절하는 *프로퍼티 시트*이다. 이것은 컴포넌트 기반의 개발 도구의 필수적인 부분이다. 왜냐하면 디자인 시점에 프로퍼티를 설정하는 것은 컴포넌트의 초기 상태를 설정하는 방법이기 때문이다.

그림 8-3 : 빈 박스의 구성요소들

가령, 여러분은 프로퍼티 시트내에 새로운 이름을 입력하는 것에 의해서 빈 박스에서 사용되는 패널의 name 속성을 재설정할 수 있다. name 속성을 변경하는 것은 단순하다-단지 텍스트 필드내에 문자열을 편집한다.

그러나 속성들은 반드시 문자열이지 않다. 이들은 자바 플랫폼에서 지원하는 모든 타입의 값일 수 있다. 사용자들이 모든 타입의 속성에 대한 값들을 설정할 수 있게 끔 하기 위해 빌더 도구는 특별한 프로퍼티 에디터(Property editor)에 접근할 수 있다. (프로퍼티 에디터는 빌더에서 제공하거나 빈 개발자에 의해서 제공된다. 여러분은 여러분 자신의 프로퍼티 에디터를 작성하는 방법에 대해 이 장의 뒤에서 보게 될 것이다.)

작동중인 단순한 프로퍼티 에디터를 살펴보자, 다음과 같이 빈 박스의 배경 색을 재설정해 보자:

1. 프로퍼티 시트내의 “background”라고 표기된 박스 내부를 클릭한다. 이것은 그림8-4에서 보여지는 것처럼 Color 형의 객체를 편집하기 위한 프로퍼티 에디터를 제공한다.
2. 첫번째 박스내에 새로운 숫자값(0에서 255사이)를 입력하거나 에디터내에 선택 박스를 통해 새로운 색을 선택한다.

그림8-4 : 속성을 설정하기 위해 프로퍼티 에디터 사용하기

여러분은 즉시 배경 색이 변경되는 것을 보게 될 것이다.

빈박스에서 빈 사용하기

빈을 사용하는 것은 어렵지 않다.

1. 툴 박스에서 빈을 선택한다.(커서가 십자선 모양으로 변할 것이다.)
2. 빈을 옮기고자 하는 빈 박스 윈도우내의 위치를 클릭한다.

사용자 인터페이스가 아주 독창적이지는 않다. 여러분이 선택한 빈에 대한 실마리를 제공하지 않을 뿐만 아니라 일단 시작한 후에 해당 기능을 취소할 수도 없다.

일단 여러분이 빈 박스 윈도우내에 빈을 가지고 있다면 이를 선택하기 위해 클릭하라. 평행선의 경계는 빈이 선택됐음을 나타낸다(그림8-5참조 – 아래에 있는 버튼이 선택되었다.)

그림 8-5 : 선택된 빈 대 선택되지 않은 빈

여러분은 다음에 의해서 선택된 빈들을 빈 박스 윈도우 주변으로 이동할 수 있다:

1. 마우스 커서를 경계로 이동한다.(여러분은 다양한 지점으로 시도할 것이다 – 여러분은 그림8-6에서 처럼 커서가 4개의 머리를 갖은 화살 모양으로 변할 때 올바른 위치라는 것을 알게 될 것이다.)
2. 빈을 새로운 위치로 끌어다 놓는다.

그림 8-6 : 이동 준비중인 빈

유사하게, 여러분은 커서를 코너들 중의 하나로 이동하는 것에 의해서 빈들의 크기를 다시 조정할 수 있다. (양방향 화살표로 변경될 것이다.) 그리고 빈의 경계를 끌어당겨 여러분이 원하는 대로 크기를 조절할 수 있다.

팁 : 빈 박스 윈도우상에 잘못 위치시킨 빈을 제거하기 위해

1. 빈을 선택한다.
2. 메뉴 옵션에서 Edit -> Cut을 선택한다.

빈박스에서 단순한 응용 프로그램 작성하기

우리는 아 장의 뒤에서 빈 박스 같은 빌더 툴에서 빈이 제공하는 속성이 무엇이고 어떤 이벤트가 트리거되는 지를 알 수 있는 방법에 대해 설명한다. 우리는 여러분에게 모든 작업을 보여 줄 것이다. 이를 위해 져글러 빈에서 보여주는 곡예하는 문자를 시작하고 멈추는 버튼을 갖는 프로그램을 작성할 것이다.(그림8-7은 이 프로그램의 결과이다.)

1. 빈 박스 윈도우에 두개의 “ExplicitButton” 빈을 추가한다.
2. 레벨 속성을 하나는 Start로 다른 하나는 Stop으로 설정한다.
3. 져글러 빈을 빈 박스에 추가한다. 져글러 빈은 바로 져글링을 시작한다는 것을 주

목하라.

그림8-7 : Start/Stop 져글러 응용 프로그램을 위한 빈 박스 윈도우

저글러가 이미 져글링을 하고 있기 때문에 먼저 "Stop" 버튼을 조작한다.

1. "Stop" 버튼을 선택한다.
2. Edit -> Events -> button push(그림 8-8 참조)을 선택한다. 해당 하위 메뉴로부터 actionPerformed 이벤트를 선택한다.

그림8-8 : Stop/Start 져글러 응용 프로그램의 Stop 버튼 윈도우에 대해 가능한 이벤트들

이제, 빈 박스 윈도우상에서 마우스 커서를 이동할 때 여러분은 Stop 버튼으로부터 뻣어나오는 붉은 선을 보게 될 것이다.(그림 8-9 참조)

그림8-9 : 이벤트 설정(hookup)을 위한 마커

이제, 저글러 빈을 선택하자. 여러분은 actionPerformed 이벤트에 대한 설정을 원하는 메소드를 선택할 수 있는 그림8-10에서 보여지는 것과 같은 윈도우를 보게 될 것이다.

우리는 져글링을 멈추길 원한다. 그래서 단지 그림 8-10에서 보여지는 다이얼로그 박스내에서 stopJuggling을 선택한다. OK를 클릭한 다음, 빈 박스는 필요로 하는 적합한 코드를 생성해 낼 것이다.(빈 박스가 저장되어 있는 위치아래 tmp\sunw\beanbox 디렉토리 내부를 살펴보라.)

그림8-10 : 목표 메소드 다이얼로그

다음은 자동 생성된 코드의 내용이다:

```
/ Automatically generated event hookup file.

package tmp.sunw.beanbox;
import sunw.demo.juggler.Juggler;
```

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class ___Hookup_16668274e7 implements java.awt.event.ActionListener, java.io.Serializable {

    public void setTarget(sunw.demo.juggler.Juggler t) {
        target = t;
    }

    public void actionPerformed(java.awt.event.ActionEvent arg0) {
        target.stopJuggling(arg0);
    }

    private sunw.demo.juggler.Juggler target;
}
```

주어진 아답터 클래스가 컴퓨터가 생성한 보기 낯설은 이름이라는 것에 주목하라.

중요한 점은 actionPerformed **메소드가** Juggler **클래스의** stopJuggling **메소드를 호출한다는 것이다. 따라서** "Stop" **버튼이 실제 져글러를 멈춘다.**

빈박스의 상태 저장과 재생

여러분이 기대했던 것처럼 빈 박스는 여러분의 빈 박스 상태를 저장하기 위해 객체 연속화를 사용한다.

노트 : 빌더 툴이 빈의 상태(현재 프로퍼티 설정처럼)를 저장하는 기능을 지원하는 것은 필수적이다. 따라서 현재의 상태를 연속화하도록 여러분의 빈을 디자인해야 한다.(1**권의** 12 **장 참조)**

응용 프로그램의 상태를 저장하기 위해 여러분은 빈 박스내에서 테스트를 해야 한다. File -> Save **를 선택하고 원하는 대로 파일 다이얼로그 박스를 채운다. 빈 박스의 상태를 재생하기 위해서는 빈 박스의 모든 빈들을 해제한 다음**(File -> Clear) **이전 응용 프로그램을 다시 로드하기위해** File -> Open **을 선택한다.**

빈박스에서 애플릿 작성하기

여러분의 빈 박스 실험을 전개하기 위해, 여러분은 다른 사람들이 부라우져나 애플릿 뷰어 내에서 수행할 수 있도록 애플릿을 생성할 수 있다.

1. File -> Make Applet**을 선택한다.**
2. **나타난 다이얼로그 박스(** 그림8-11)**에 새로운** JAR **파일 이름을 제공하거나 기본 값을 수용할 수 있다.**

그림8-11 : 애플릿 생성 다이얼로그

모든 것은 끝났다. 빈 박스는 필요한 HTML **코드(**myApplet.html**이 기본 이름이다.)를 생성해 낼 것이다. 여러분은 자바 가능한 부라우져와 애플릿 뷰어(** 그림 8-12 **)에서** HTML **파일을 로드 할 수 있다.**

그림 8-12 : 애플릿 뷰어에서 애플릿으로서 연결된 빈들

툴박스에 빈들 추가하기

임의의 빈을 빌더 도구에서 사용 가능하게 만들기 위해서는 빈 코드에서 사용되는 모든 클래스 파일들을 JAR **파일내에 패키지 해야 한다. 이전에 보았던 애플릿을 위한** JAR **파일들과 달리 빈을 위한** JAR **파일은 매너패스트(**manifest**) 파일을 필요로 한다. 매너패스트 파일은 아카이브내에 어느 클래스 파일들이 빈인지를 명시하고 툴박스내에 포함돼야 한다. 가령, 다음은** ImageViewerBean**을 위한 매너패스트 파일인** ImageViewerBean.mf **이다.**

```
Name: ImageViewerBean.class
Java-Bean: True
```

만일 여러분의 빈이 복수 클래스 파일들을 포함하고 있다면 여러분은 단지 매너패스트에 이들 클래스 파일들이 빈이고 툴박스에서 출력되길 원한다는 것을 명시하면 된다.

JAR **파일들을 만들기 위해 , 다음의 단계들이 있다:**

1. **매너패스트 파일을 편집한다.**

2. 디렉토리내에 필요한 모든 클래스 파일들을 모은다.
3. 다음처럼 jar 도구를 수행한다.

   jar cfm Jarfile ManifestFile *.class

   가령,

   jar cfm ImageViewerBean.jar ImageViewerBean.mf *.class

또한 여러분은 JAR 파일에 GIF 파일 같은 다른 아이템들을 추가할 수 있다.

팁 : 만일 여러분이 필요한 JAR 파일을 생성하는 데 make 파일을 더 선호한다면 몇몇 예제 make(.mk) 파일들을 bdk 아래 demo 디렉토리에서 살펴 볼 수 있다.

마지막으로, 빈 박스의 툴 박스가 여러분의 빈을 포함하고 있는 JAR 파일을 인식하도록 하기위해 JAR 파일을 BDK의 jars 하위 디렉토리에 복사하거나 빈 박스 윈도우의 File -> Load Jar 를 선택해야 한다.

빈으로 이미지 뷰어 응용 프로그램 개발

우리는 여러분들이 비쥬얼 베이직 ImageViewer 프로그램과 동일한 기능을 제공하는 두개의 프로그램을 사용할 수 있기에 충분한 빈 박스 기술을 보여주었다.(이들 프로그램은 CD-ROM에 포함되어 있다.) 이러한 빈들이 작동하는 걸 보기위해 ImageViewerBean과 FileNameBean JAR 파일들을 File->Load Jar를 선택하거나 bdk 디렉토리의 jar 하위 디렉토리에 JAR 파일을 복사하여 빈 박스에 추가해야 한다.

노트 : 여러분은 이미 IamgeViewerBean의 코드를 살펴보았다. FileNameBean의 코드는 보다 복잡하다. 우리는 이 장의 뒤에서 보다 깊이 분석해 볼 것이다. 지금까지 여러분이 알아야 할 모든 것은 파일의 이름을 입력할 수 있는 표준 파일 개발 다이얼로그 박스를 개방하는 부호를 클릭하는 것이다.

만일 여러분이 ImageViewerBean을 추가했다면 그림 8-13을 볼 수 있을 것이다. 여러분은 GIF나 JPEG 파일을 위한 파일 이름을 선택할 수 있다. 일단 선택을 하면 ImageViewerBean 은 자동적으로 빈을 출력한다.

그림 8-13 : 작동중인 ImageViewerBean

노트 : 만일 여러분이 그림 8-13의 오른편의 프로퍼티 시트를 살펴본다면 autoscrolls **과** requestFocusEnabled **처럼 많은 수의 신비한 프로퍼티들을 발견할 것이다. 이 들 속성은** Jpanel **상위 클래스로부터 상속받은 것이다. 여러분은 이장의 뒤에서 이들을 프로퍼티 시트에서 제거하는 방법을 보게 될 것이다.**

주의 : 만일 여러분이 빈 박스를 구동시킨 쉘 윈도우를 살펴본다면 다음과 같은 신기한 메세지들을 보게 될 것이다.

Warning : Can' t fine public property editor for property "nextFocusableComponent". Skipping.

빈 박스에서 자동 속성 감지는 nextFocusableComponent**라 불리는 속성을 믿도록 속인다. 그러나 프로퍼티 시트에 컴포넌트 값을 출력하는 방법을 알지 못한다. 여러분은 이 장의 뒤에서 빈 박스에게 빈의 진정한 속성들이 무엇인지 알려주는 방법을 보게 될 것이다.**

비쥬얼 베이직 프로그램의 기능을 모방하기 위해, 져글러에서 했던 것처럼 이벤트들을 훅업 (hook up)**하지 않는다. 대신에 우리는 빈의** filename **속성을 함께 연결한다. 아이디어는** FileNameBean**의** filename **속성을** ImageViewerBean**의** filename **속성이 현재 값을 즉시 반영하도록 변경하는 것이다. (** 물론, **이것은 실제적으로** PropertyChange **이벤트를 통해서 발생한다. 우리는 이 장의 뒤에서 이러한 종류의 이벤트들에 대해서 논의할 것이다.) 반대로 해당 파일 이름이 변하면** ImageViewerBean**의 결과는** VB **프로그램과 동일하게 해당 이미지를 바로 출력한다.**

노트 : 우리는 빈 박스가 작동한다고 가정하고 설명한다. 빈 박스의 가장 최신 버전을 사용하라. 몇몇 버전은 이미지를 갱신하지 않고, 몇몇 버전은 프로퍼티 시트를 갱신하지 않는다. 그리고 불운하게도 몇몇 버전은 둘 다 지원하지 않는다.

각각의 빈들의 프로퍼티를 훅업하기 위해:

1. FileNameBean을 선택한다.
2. Edit->Bind 속성을 선택한다.
3. 팝업된 결과 다이얼로그(그림 8-14)에서 filename 속성을 선택한다.
4. OK를 클릭한다.

그림 8-14 : FileNameBean을 위한 바인드 PropertyNameDialog 박스

이제, 마법의 붉은 선이 나타날 것이다. 그리고 여러분은 이 선을 ImageViewerBean에 도달하도록 한다.

붉은 선이 ImageViewerBean에 도달할 때 클릭한다. 그러면 빈 박스는 다른 다이얼로그 박스(그림 8-15)를 팝업할 것이다. fileName을 클릭하고 "OK"를 선택한다.

그림 8-15: ImageViewerBean을 위한 PropertyNameDialog 박스

이 시점에서, 두개의 filename 프로퍼티들이 훅업되었다. 그래서 여러분은 생략기호(...)를 클릭하고 GIF 파일의 이름을 선택함으로써 이미지를 갱신하기 위해 FileNameBean을 사용할 수 있다.(그림8-16 참조)

그림 8-16 : filename 프로퍼티의 훅업 결과

빈 속성과 이벤트를 위한 명명 규칙

우리는 먼저 여러분의 빈을 구성할 때 확장할 최상위 빈 클래스가 없다는 것을 강조하고 싶다. 비쥬얼 빈은 직접적이거나 간접적으로 Component 클래스를 확장한다. 그러나 비쥬얼 빈이 아니면 비쥬얼 디자인 도구에서 조작할 수 있는 임의의 클래스이다. 디자인 도구는 클래스의 빈 상태를 결정하기 위해 상위 클래스를 참조하지 않는다. 그러나 해당 클래스의 메소드의 이름을 분석한다. 이러한 분석을 가능하게 하기위해 빈을 위한 메소드 이름은 어떤 패턴을 따라야 한다.

노트 : java.beans.Beans 클래스가 존재하지만 이 클래스의 모든 메소드는 정적 메소드이다. 따라서 이 클래스를 확장하는 것은 의미가 없다. 간혹 확장하는 것을 보게 되겠지만. 분명 빈은 Beans와 Component를 확장할 수 없다. 이러한 접근 방법은 비쥬얼 빈들에는 해당하지

않는다. 사실 Beans 클래스는 가령, 빌더 도구가 디자인 작업 중인지, 실행 중인지를 검사하는 것처럼 빌더 도구들에 의해서 호출되어지도록 설계된 메소드들을 포함하고 있다.

비쥬얼 베이직이나 델파이 같은 비쥬얼 디자인 환경들을 위한 다른 언어들은 이러한 개념을 직접적으로 표현하기 위해 "Property"나 "Event"같은 특별한 키워드를 가지고 있다. 자바 명세의 설계자는 비쥬얼 프로그래밍을 지원하기 위해 언어에 키워드를 추가하지 않기로 결정하였다. 그러므로 이들은 빌더 도구에서 빈의 속성이나 이벤트들을 알기 위해 빈을 분석할 수 있도록 다른 대안을 필요로 했다. 실제 두 개의 다른 메커니즘이 존재한다. 만일 빈 작성자가 속성과 이벤트들을 위한 표준 명명 패턴을 사용한다면 빌더 도구는 빈이 제공하는 속성과 이벤트들을 이해하기 위해 리플렉션 메커니즘을 사용한다. 다른 대안으로 빈 작성자는 빌더 도구에게 빈의 속성들과 이벤트들에 대해 말하기 위해 빈 정보 클래스를 제공할 수 있다. 우리는 사용하기 쉽기 때문에 명명 패턴을 사용하여 시작할 것이다. 여러분은 이 장의 뒤에서 빈 정보 클래스를 제공하는 방법을 보게 될 것이다.

노트 : 비록 문서는 이러한 표준 명명 패턴을 "디자인 패턴"이라고 부르지만 이들은 실제 명명 규칙이고 객체지향 프로그래밍에서 사용되는 디자인 패턴과는 상관이 없다.

속성을 위한 명명 패턴은 단순하다: 타입 X의 읽기/쓰기 속성에 해당하는 다음 메소드들의 임의의 쌍이다.

```
public X getPropertyName()
pubic void setPropertyName(X x)
```

예를 들면, ImageViewerBean에서는 (출력할 파일 이름을 위해) 오직 하나의 read/write 속성만이 존재한다:

```
public String getFileName();
public void setFileName(String f)
```

만일 여러분이 get 메소드를 가지고 있고 연관된 set 메소드가 없다면 여러분은 읽기만 가능한 속성을 정의한 것이다.

여러분의 메소드 이름들을 위해 사용할 대소문자 패턴에 주의하라. 자바빈 명세의 디자이너들은 우리의 예제에서 비록 get과 set 메소드가 대문자F(getFileName , setFileName)를 포함하고 있지만 속성의 이름이 소문자 f로 시작하는 fileName이 되도록 결정하였다. 빈 분석가

(analyzer)는 속성 이름을 연역하기 위해 decapitalization 이라 불리는 과정을 수행한다. (즉, get이나 set뒤의 첫번째 문자를 소문자로 변환한다.) 이 과정은 프로그래머에게 보다 자연스러운 메소드와 프로퍼티 이름을 제공한다.

노트 : 여러분이 생성한 get과 set 메소드는 사적 데이터 필드를 단순히 get과 set하는 것 이상의 작업을 수행한다. 모든 자바 메소드처럼 이들은 독자적인 행동을 수행할 수 있다. 가령, ImageViewerBean 클래스의 setFileName 메소드는 filename 데이터 필드의 값을 설정할 뿐만 아니라 파일을 개발하고 이미지를 로드한다.

노트 : 비쥬얼 베이직에서 속성들은 get과 set 메소드로부터 나온다. (델파이는 read와 write를 사용한다.) 그러나 이들 언어는 모두 컴파일러가 메소드 이름들을 분석하여 프로그래머의 의도를 추측할 필요가 없도록 Property 키워드를 가지고 있다. 그리고 이들 언어에서는 키워드를 사용하는 것은 다른 이점이 있다: = 연산의 왼쪽에 속성 이름을 사용하는 것은 자동적으로 set 메소드를 호출한다. 표현식에서 속성 이름을 사용하는 것은 자동적으로 get 메소드를 호출한다. 예를 들면, VB에서 여러분은 아래 표현 대신에

```
imageBean.setFileName(“corejava.gif”);
```

다음 표현을 사용할 수 있다.

```
imageBean.setFileName = “corejava.gif”;
```

---

get/set 이름 패턴에는 한가지 예외가 있다. 부울리언 값을 갖는 속성들은 다음의 예제처럼 is/set 이름 패턴을 사용할 수 있다.

```
public boolean isPropertyName()
public void setPropertyName(Boolean b)
```

가령, 애니메이션은 두개의 메소드를 갖는 running 속성을 갖을 수 있다.

```
public boolean isRunning()
public void setRunning(Boolean b)
```

setRunning 메소드는 애니메이션을 시작하고 멈춘다. isRunning 메소드는 현재의 상태를 알려준다.(그러나 져글러 빈은 이 같은 속성을 갖지 않는다.)

이벤트들에 대한 이름 패턴은 보다 단순하다. 빈 빌더 환경은 여러분이 이벤트 리스너를 추가하거나 제거하는 메소드를 제공할 때 여러분의 빈은 자동으로 이벤트들을 추정할 것이다. 여러분의 빈이 EventNameEvent 타입의 이벤트들을 생성한다고 가정해 보자.(모든 이벤트들은 Event로 끝난다.) 그러면 리스너 인터페이스는 EventNameListener를 호출해야 한다. 그리고 리스너를 추가하고 제거하는 메소드들이 호출돼야 한다.

```
public void addEventNameListener(EventNameListener e)
public void removeEventNameListener(EventNameListener e)
```

만일 여러분이 ImageViewerBean의 코드를 살펴보았다면 노출되는 이벤트가 없다는 것을 알게 될 것이다. 나중에 우리는 TimerEvent 객체를 생성하고 이벤트 리스너들을 관리기 위해 다음 메소드들을 갖고 있는 타이머 빈을 보게 될 것이다:

```
public void addTimerListener(TimerListener e)
public void removeTimerListener(TimerListener e)
```

노트 : 만일 사용자들이 프로퍼티 시트를 조작하길 원하는 프로퍼티나 이벤트에 해당하지 않는 일련의 get과 set 메소드들을 가지고 있다면 어떻게 해야 할 까? 물론 여러분은 여러분의 클래스내에서 메소드들의 이름을 재명명하여 이러한 상황을 피할 수 있다. 그러나 만일 여러분이 다른 클래스를 확장한다면 상위 클래스로부터 메소드 이름들을 상속해야 한다.
가령, 이러한 상황은 여러분의 빈이 JPanel을 확장할 때 발생한다 – 많은 수의 흥미없는 속성들이 프로퍼티 시트에 나타난다. 이것은 명명 패턴들이 실제 프로퍼티들을 디자인하는 데 좋지않은 방법이라는 것을 보여준다. 여러분은 이 장의 뒤에서 프로퍼티에 해당하는 메소드가 무엇인지 정확히 명시하는 빈 정보를 제공하여 자동 프로퍼티 발견 과정을 오버라이드할 수 있는 방법을 보게 될 것이다.

빈 속성 타입

복잡한 빈은 사용자들이 디자인 시점에 프로퍼티를 설정하거나 실행중에 프로퍼티를 얻기위한 빌더 도구내에 많은 다른 종류의 속성들을 갖을 것이다. 또한 표준 이벤트와 커스텀 이

벤트간의 트리거를 할 수 있다. 프로퍼티들은 여러분이 ImageViewerBean과 FileNameBean에서 보았던 filename 속성만큼 단순하거나 색상 값처럼 복잡하거나 심지어 데이터 포인터의 배열일 수 있다 – 우리는 이장의 뒤에서 이들 경우를 모두 만난다. 더욱이, 프로퍼티들은 이 절에서 보게 될 것처럼 이벤트를 발생할 수 있다.

여러분의 빈들의 속성들을 올바로 모으는 것은 아마 빈을 작성하는 데 있어 가장 복잡할 것이다. 왜냐하면 모델이 아주 풍부하기 때문이다. 다양한 예제를 통해 우리가 소개하는 것처럼 자바빈 명세는 4개 타입의 속성이 가능하다.

단순 속성들

단순 속성은 문자열이나 숫자 같은 단일 값을 취한다. ImageViewer의 fileName 속성은 단순 속성의 예이다. 단순 속성들은 프로그램하기 쉽다: 단지 우리가 앞서 설명했던 set/get 명명 규칙을 사용한다. 예를 들면, 만일 여러분이 예제8-1의 코드를 살펴본다면 단순 문자열 속성을 구현한 것을 볼 수 있다.

```
public void setFileName(String f)
{   fileName = f;
    image = Toolkit.getDefaultToolkit().getImage(fileName);
      …// use media tracker to wait for image loading
     repaint();
}

public String getFileName()
{   return fileName;
}
```

자바 빈 명세가 관련이 있는 한 우리는 해당 빈의 읽기만 하는 속성을 가지고 있다. 왜냐하면 우리는 클래스내에 setPreferredSize 를 갖지 않는 다음 서명을 갖는 메소드를 갖고 있다.

```
public Dimension getPreferredSize()
```

여러분은 일반적으로 프로퍼티 시트에서 디자인 시 읽기만 하는 속성들을 볼 수 있을 것이다.

팁: 우리가 작업하고 있는 빈박스의 버전에서, 여러분은 사적 데이터 멤버를 널이 아닌 값으로 극명하게 초기화해야 한다. 그렇지 않으면 빈 박스는 단순이 이들을 출력하지 않고 무엇인가 철자를 잘못 표기했는가에 대한 의문을 여러분에게 남겨놓는다.

```
public String getProp() { return prop; }
public void setProp(String P) { prop = p; }
….
private String prop;
        // initialize with null – BeanBox wont' t display prop
```

대신에, 빈 박스가 빈을 로드할 때 컨솔 윈도우에 에러 메시지를 출력할 것이다. 만일 프로퍼티가 프로퍼티 시트에 보여지지 않는다면 이러한 에러 메시지들을 검사해 보아라.

색인된 속성들

색인된 속성들은 배열을 가져오고 설정한다. 차트 빈은 데이터 포인터들에 대한 색인된 속성을 사용한다. 색인된 속성을 갖고 여러분은 두 쌍의 get과 set 메소드를 제공한다: 하나는 배열을 위한 다른 하나는 개별 항목들을 위한 메소드이다.

```
X[] getPropertyName()
void setPropertyName(X[] x)
X getPropertyName(int I)
void setPropertyName(int i, X x)
```

다음은 이 장의 뒤에서 여러분이 보게 될 차트 빈에서 사용하는 색인된 속성의 예제이다.

```
public double[] getValues() { return values; }
public void setValues(double[] v) { values = v; }
public double getValues(int i) { return values[i]; }
public void setValues(int i , double v) { values[i] = v; }
….
private double[] values;
```

개별적인 배열 항목들을 설정하는 get/set 함수들은 i 가 합법적인 범위 내에 있다고 가정한다. 빌더 도구는 0보다 적은 색인이나 배열의 길이보다 큰 i를 호출하지 않는다. 특히,

```
setPropertyName( int i , X[] x)
```

이 메소드는 배열의 증가(grow)를 위해 사용할 수 없다. 배열의 증가를 위해 여러분은 직접 새로운 배열을 생성하고 이를 다음 메소드에 전달해야 한다.

```
setPropertyName( X[] x)
```

그러나 실제적으로 get과 set 메소드는 단지 빌더 환경에 의해서가 아니라 여전히 프로그래밍적으로 호출될 수 있다. 그러므로 결국 몇몇 에러 검사를 추가하는 것이 최선의 방법이다. 가령, 다음은 ChartBean 클래스내에서 실제 우리가 사용하는 코드이다.

```
public double getValues(int i)
{       if( 0 <= i && i < values.length) return values[i];
        return 0;
}

public void setValues(int i, double value)
{       if( 0 <= i && i < values.length) values[i] = value;
}
```

노트 : 현재 시점에서, 빈박스의 프로퍼티 시트는 색인 속성들을 지원하지 않는다. 즉, 이들 속성들은 프로퍼티 시트상에 보여지지 않는다. 그리고 여러분은 빌더내에서 이들을 설정할 수 없다. 여러분은 이장의 뒤에서 배열을 위한 속성 편집기를 제공함으로써 이러한 제약을 극복하는 방법을 보게 될 것이다.

바운드 속성들

바운드 속성들은 그들의 값이 변하는 것을 관심있는 리스너에게 말한다. 가령, FileNameBean의 filename 속성은 바운드 속성이다. 파일 이름이 변경될 때, ImageViewerBean은 자동적으로 이를 알려주고 새로운 파일을 로드한다.

바운드 속성을 구현하기 위해, 여러분은 두개의 메커니즘을 구현해야 한다.

1. 속성 값이 변경할 때마다, 빈은 PropertyChange 이벤트를 모든 등록된 리스너에게 보내야 한다. 이러한 변화는 set 메소드가 호출될 때나 텍스트를 편집하거나 파일을 선택할 때처럼 프로그램 사용자가 액션을 취할 때 발생한다.
2. 관심있는 리스너들에게 이들을 등록하기 위해, 빈은 다음 두개의 메소드들을 구현해야 한다:

   ```
   void addPropertyChangeListener(PropertyChangeListener listener)
   void removePropertyChangeListener(PropertyChangeListener listener)
   ```

java.beans 패키지는 PropertyChangeSupport라 불리는 편리한 편리한 패키지를 가지고 있다. 이 패키지는 여러분을 위해 리스너를 관리한다. 이 편리한 클래스를 사용하기 위해, 여러분

의 빈은 다음처럼 해당 클래스의 데이터 필드를 가져야 한다.

```
private PropertyChangeSupport changeSupport = new PropertyChangeSupport(this);
```

여러분은 이 객체에게 프로퍼티의 변경을 리스너에게 추가하고 삭제하는 일을 위임한다.

```
public void addPropertyChangeListener
    (PropertyChangeListener listener)
{   changeSupport.addPropertyChangeListener(listener);
}

public void removePropertyChangeListener
    (PropertyChangeListener listener)
{   changeSupport.removePropertyChangeListener(listener);
}
```

속성의 값이 변할 때마다 PropertyChangeSupport 객체의 changeSupport 메소드는 한 이벤트를 모든 등록된 리스너들에게 전달하기 위해 사용한다. 이 메소드는 3개의 전달인자를 갖고 있다: 속성의 이름, 이전 값, 새로운 값. 예를 들면

```
changeSupport.firePropertyChange ( "filename", oldValue , newValue);
```

여기서 값들(value)은 객체여야 한다. 만일 속성 타입이 객체가 아니면 여러분은 포장 객체(wrapper Object)를 사용해야 한다. 예를 들면,

```
changeSupport.firePropertyChange ( "running", new Boolean(false) , new Boolean(true) );
```

팁 : 만일 여러분의 빈이 최종적으로 JComponent 클래스를 확장한 임의의 스윙 클래스를 확장한다면 addPropertyChangeListener와 removePropertyChangeListener 메소드를 구현할 필요가 없다. 이들 메소드들은 이미 JComponent 상위 클래스내에서 구현되었다.

속성 변경를 리스너에게 알리기 위해, JComponent 상위 클래스의 firePropertyChange 메소드를 호출하면 된다:

```
firePropertyChange( "propertyName", oldValue, newValue);
```

여러분의 편의를 위해, 이 메소드는 Boolean, byte, char, double, float, int, long 그리고 short 타입을 위해 오버로드 되었다. 만일 oldValue와 newValue가 이들 타입들에 포함되면 포장 객체를

사용할 필요가 없다.

속성 값이 변할 때 이를 감지하길 원하는 다른 빈은 PropertyChangeListner 인터페이스를 구현해야 한다. 이 인터페이스는 오직 하나의 메소드를 포함하고 있다.

```
void propertyChange(PropertyChangeEvent event)
```

propertyChange 메소드내의 코드는 속성 값이 변할 때마다 트리거된다. 물론, 여러분은 이벤트를 생성해내는 빈의 속성 변경 리스너들에게 감지받을 곳(recipient)을 추가해야 한 것을 제공한다. propertyChangeEvent 객체는 속성의 이전 값과 새로운 값을 캡슐화해야 한다. 이 값들은 다음을 통해 얻을 수 있다.

```
Object oldValue = event.getOldValue();
Object newValue = event.getNewValue();
```

만일 속성 타입이 클래스 타입이 아니면 반환된 객체들은 일반 포장 타입이다. 가령, 만일 boolean 속성이 변경되면 Boolean이 반환되고 여러분은 booleanValue 메소드로 부울리언 값을 추출할 필요가 있다.

따라서, 리스닝 객체는 다음 모델을 따라야 한다"

```
class Listener implements PropertyChangeListener
{       public Listener()
        {       bean.addPropertyListener(this);
        }
        void propertyChange(propertyChangeEvent event)
        {       Object newValue = event.getNewValue();
                …
        }
        …
}
```

여러분은 FileNameBean에서 파일 이름이 변경될 때 ImageViewerBean이 어떻게 이를 감지하는 지 궁금할 것이다. 최종적으로, ImageViewerBean 소스 코드에 리스너 메소드는 없다. 이 경우, 빈 박스는 FileNameBean에 속성 변경 리스너로서 그 자체를 등록한다. 그리고 해당 propertyChange 리스너 메소드에서 ImageViewerBean의 setFileName 메소드를 호출한다.

예제8-2는 FileNameBean의 완전한 소스 코드이다. FileNameBean이 JPanel 클래스를 확장했기 때문에 우리는 propertyChangeSupport 객체를 극명하게 사용할 필요가 없다. 대신에, 우리는

속성 변경 리스너들을 관리하는 JPanel 클래스의 기능에 의존한다.

예제 8-2 : FileNameBean.java

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.io.*;
import javax.swing.*;

public class FileNameBean extends JPanel
   implements Serializable
{  public FileNameBean()
   {  dialogButton = new JButton("...");
      nameField = new JTextField("");

      chooser = new JFileChooser();
       chooser.setCurrentDirectory(new File("."));

       chooser.setFileFilter(
          new javax.swing.filechooser.FileFilter()
          {  public boolean accept(File f)
             {  String name = f.getName().toLowerCase();
                return name.endsWith("." + defaultExtension)
                      || f.isDirectory();
              }
             public String getDescription()
             {  return defaultExtension + " files";
              }
          });

       setLayout(new GridBagLayout());
      GridBagConstraints gbc = new GridBagConstraints();
       gbc.weightx = 100;
       gbc.weighty = 100;
       gbc.anchor = GridBagConstraints.WEST;
       gbc.fill = GridBagConstraints.BOTH;
```

```
    add(nameField, gbc, 0, 0, 1, 1);
     dialogButton.addActionListener(
        new ActionListener()
       {  public void actionPerformed(ActionEvent evt)
          {  showFileDialog();
          }
        });
     nameField.setEditable(false);
    gbc.weightx = 0;
    gbc.anchor = GridBagConstraints.EAST;
    gbc.fill = GridBagConstraints.NONE;
   add(dialogButton, gbc, 1, 0, 1, 1);
 }

public void add(Component c, GridBagConstraints gbc,
   int x, int y, int w, int h)
{  gbc.gridx = x;
     gbc.gridy = y;
    gbc.gridwidth = w;
    gbc.gridheight = h;
     add(c, gbc);
 }

public void showFileDialog()
{  int r = chooser.showOpenDialog(null);
    if(r == JFileChooser.APPROVE_OPTION)
    {  String name
          = chooser.getSelectedFile().getAbsolutePath();
       setFileName(name);
     }
 }

public void setFileName(String newValue)
{  String oldValue = nameField.getText();
    firePropertyChange("fileName", oldValue, newValue);
     nameField.setText(newValue);
```

```
    }

   public String getFileName()
    {   return nameField.getText();
    }

   public Dimension getMinimumSize()
   {   return new Dimension(XMINSIZE, YMINSIZE);
    }

   public String getDefaultExtension()
    {   return defaultExtension;
    }

   public void setDefaultExtension(String s)
    {   defaultExtension = s;
    }

  private static final int XMINSIZE = 200;
  private static final int YMINSIZE = 20;
   private JButton dialogButton;
   private JTextField nameField;
   private JFileChooser chooser;
    private String defaultExtension = "gif";
}
```

java.beans.PropertyChangeListener

- void propertyChange(PropertyChangeEvent event)
  속성 변경 이벤트가 발생할 때 호출된다.
  전달인자: event **속성 변경 이벤트**

java.beans. PropertyChangeSupport

- PropertyChangeSupport(Object sourceBean)
  PropertyChangeSupport **객체를 구성한다**.
  전달인자: sourceBean **속성 변경의 소스인 빈(일반적으로** this **임)**

- void addPropertyChangeListener(PropertyChangeListener listener)
  바운드 속성을 대해 관심있는 리스너를 등록한다.
  전달인자: listener **바운드 속성내에서 변화가 감지되길 원하는 객체**

- void removePropertyChangeListener(PropertyChangeListener listener)
  바운드 속성을 대해 이전에 등록된 관심있는 리스너를 제거한다.
  전달인자: listener **리스너들의 목록에서 제거될 객체**

- void firePropertyChange(String propertyName, Object oldValue, Object newValue)
  등록된 리스너에게 PropertyChangeEvent**를 전달한다.**
  전달인자: propertyName **프로퍼티 이름**
  oldValue **이전 값**
  newValue **새로운 값**

java.beans. PropertyChangeEvent

- PropertyChangeEvent(Object source , String propertyName, Object oldValue, Object newValue)
  새로운 PropertyChangeEvent **객체를 구성한다.**
  전달인자: object **속성을 위한 빈 소스**
  propertyName **프로퍼티 이름**
  oldValue **이전 값**
  newValue **새로운 값**

- Object getNewValue()
  속성의 새로운 값을 반환한다.
- Object getOldValue()
  속성의 이전 값을 반환한다.
- String getPropertyName()
  속성의 이름을 반환한다.

javax.swing.JComponent

- void addPropertyChangeListener(String propertyName, PropertyChangeListener listener)
  바운드 속성에 대해 관심있는 리스너를 등록한다.
  전달인자: propertyName **리슨할 객체에 대한 속성 이름**
  listener **바운드 속성내에서 변경을 감지하길 원하는 객체**

- void addPropertyChangeListener(PropertyChangeListener listener)
  해당 컴포넌트의 모든 바운드 속성들에 대해 관심있는 리스너를 등록한다.
  전달인자: listener **바운드 속성내에서 변경을 감지하길 원하는 객체**

- void removePropertyChangeListener(String propertyName,PropertyChangeListener listener)
  바운드 속성에 대해 이전에 등록된 관심있는 리스너를 제거한다.
  전달인자: propertyName **리슨할 객체에 대한 속성 이름**
  listener **바운드 속성내에서 변경을 감지하길 원하는 객체**

- void removePropertyChangeListener(PropertyChangeListener listener)
  해당 컴포넌트의 모든 바운드 속성들에 대해 관심있는 리스너를 제거한다.
  전달인자: listener **리스너 목록에서 제거될 객체**

- void firePropertyChange(String propertyName, Xxx oldValue, Xxx newValue)
  등록된 리스너에게 PropertyChangeEvent**를 보낸다.**
  전달인자: propertyName **프로퍼티 이름**
  oldValue **이전 값**
  newValue **새로운 값**

제약 속성들

제약 속성은 가장 관심있는 속성이다. 그리고 가장 구현하기 복잡하다. 이와 같은 속성은 예상된 변화들을 거부할 수 있는 임의의 리스너에 의해서 제약받고 이러한 변경을 이전의 설정으로 강제 복귀시킨다.

가령, 숫자를 입력받는 텍스트 필드인 IntTextBean**을 고려해 보자. 해당 숫자의 소비자는 값들에 제약을 갖을 둘 것이다. 예를 들면,** 0**과** 255**사이의 숫자이어야 한다 처럼. 이와 같은 제약은 구현하기 쉽다 :** minValue**와** maxValue**를** IntTextBean**에 추가한다. 그러나 제약은 이보다 훨씬 복잡할 수 있다. 숫자는 짝수이거나 다른 숫자에 따라 변할 수 있다. 이 경우, 사용자의 나쁜 입력으로부터 좋은 사용자의 입력을 구분할 수 있는** IntTextBean**의 단순한 속성은 없다. 대신에 빈은 소비자에게 새로운 값을 알려주고 소비자에게 이러한 변경을 거부할 수 있는 기회를 제공한다. 거부할 수 있는 속성을 *제약 속성*이라 부른다.**

우리는 이러한 개념을 범위 빈(range bean**)에 적용한다(그림** 8-17 **참조). 이 빈은 범위의 상위 바운드와 하위 바운드를 명시하기 위해 두개의** IntTextBean **필드를 포함하고 있다. (여러분**

은 사용자가 출력할 페이지들의 범위를 지정할 수 있는 프린트 다이얼로그에서 이와 같은 빈을 사용할 것이다.) 만일 사용자가 현재의 from 값보다 적은 to 값을 입력한다면(또는 from 값이 현재의 to 값보다 크다면) RangeBean은 변경을 거부한다.

그림 8-17 : 범위 빈

이 예제에서, IntTextBean은 거부가능한 이벤트의 생산자이다. 그리고 RangeBean은 거부할 수 있는 소비자이다.

제약 속성을 구성하기 위해, 여러분의 빈은 VetoableChangeListener 객체들을 등록하게 해주는 다음 두개의 메소드를 가져야 만 한다:

```
public void addVetoableChangeListener(VetoableChangeListener listener );
public void removeVetoableChangeListener(VetoableChangeListener listener );
```

속성 변경 리스너들을 관리해 주는 편리한 클래스처럼 거부 가능한 변화 리스너들을 관리하는 편리한 클래스인 VetoableChangeSupport가 존재한다. 여러분의 빈은 이 클래스의 객체를 포함해야 한다.

```
private VetoableChangeListener vetoSupport = new VetoableChangeListener(this);
```

리스너들을 추가하고 삭제하는 것은 이 객체에게 위임해야 한다. 예를 들면:

```
public void addVetoableChangeListener(VetoableChangeListener listener)
{       vetoSupport.addVetoableChangeListener(listener);
}
public void removeVetoableChangeListener(VetoableChangeListener listener)
{       vetoSupport.removeVetoableChangeListener(listener);
}
```

팁 : JComponent 클래스는 재약 속성들을 위한 몇몇 지원을 가지고 있다. 그러나 바운드 속성들을 위한 것만큼 확장하지는 못한다. JComponent클래스는 각 속성에 대한 별도의 목록이 아니라 거부가능한 변화 리스너들에 대한 단일 리스너 목록을 유지한다. 그리고 fireVetoableChange 메소드는 기본 형들을 오버로드하지 않는다. 만일 여러분의 빈이 JComponent를 확장하고 단일 제약 속성을 갖고 있다면(일반적인 경우임) , JComponent 상위

클래스를 지원하는 리스너라면 충분하다. 그리고 여러분은 별도의 VetoableChangeSupport 객체를 필요로 하지 않는다.

변경을 거부할 수 있는 객체는 VetoableChangeListener 인터페이스를 구현할 필요가 있다. 해당 인터페이스는 하나의 메소드를 포함하고 있다.

```
void vetoableChange(PropertyChangeEvent event) throws PropertyVetoException
```

이 메소드는 PropertyChangeEvent 객체를 받는다. 이 객체는 getOldValue와 getNewValue 메소드들을 사용하여 현재 값을 추출할 수 있으며 새로운 값을 제공할 수 있다.

VetoableChange가 예외상황을 추구한다는 것을 기억하라. 리스너는 PropertyVetoException을 투구하는 것에 의해서 제출된 변경을 거부한다는 것을 나타낸다. 이러한 예외상황은 빈이 새로운 값을 포기하는 이벤트를 생산하도록 강제한다.

VetoableChangeListener 인터페이스를 구현하는 방법을 보기위해 RangeBean 클래스의 코드를 살펴보자. 다음의 코드로 필요한 메소드를 구현한다.

```
public void vetoableChange(PropertyChangeEvent event)
        throws PropertyVetoException
   {   int v = ((Integer)event.getNewValue()).intValue();
      if (event.getSource() == from && v > to.getValue())
          throw new PropertyVetoException("from > to", event);
      if (event.getSource() == to && v < from.getValue())
           throw new PropertyVetoException("to < from", event);
    }
```

제약 속성들이 바운드 속성들이 되도록 강력히 추천한다. 즉, 여러분의 빈은 VetoableChangeListener 객체외에 PropertyChangeListener 객체를 추가해야 한다. 여러분은 앞 절에서 이들 메소드들을 구현하는 방법을 살펴보았다.

제약 속성 값을 갱신하기 위해, 빈은 다음 2단계 접근방법을 따른다:

1. 속성의 값을 변경하기 위한 의도(intent)를 모든 거부가능한 변화 리스너들에게 알린다. (VetoableChangeSupport 클래스의 fireVetoableChange 메소드를 사용한다.)

2. 만일 PropertyVetoException을 투구하는 거부가능한 변화 리스너들이 없다면 속성의 값을 갱신한다.
3. 모든 속성 변경 리스너에게 변화가 발생했음을 확인했음을 알린다.

다음은 IntTextBean의 기본적인 예제 코드이다.

```
public void setValue(int v) throws PropertyVetoException
  {  Integer oldValue = new Integer(getValue());
     Integer newValue = new Integer(v);
      veto.Support.fireVetoableChange("value", oldValue, newValue);
     // survived, therefore no veto
      setText("" + v);
      firePropertyChange("value", oldValue, newValue);
   }
```

모든 등록된 변경가능한 변화 리스너들이 제출된 변경에 동의한 후에 속성 값을 변경해야 한다는 것이 중요하다. 반대로, 거부가능한 변화 리스너는 결코 변화가 실제로 발생했다고 가정하지 않는다. 유일하게 신뢰가능한 방법은 변화가 실제 발생했을 때 속성 변경 리스너를 통해 아는 것이다.

예제 8-3은 거부가능한 값을 허락하는 정수 텍스트 빈에 대한 코드를 보여준다.- 이것은 1권의 IntTextField 클래스에 기반한다. 예제 8-4는 RangeBean의 코드를 보여준다. 범위 빈은 두 개의 IntTextBean 객체를 포함하고 있고 결과가 유효하지 않은 범위내에 있다면 변경을 거부한다.

범위 빈을 구동시킨 다음 to 필드의 값보다 큰 값을 from 필드에 입력하고 엔터 키를 누른다. 여러분은 from 필드의 내용이 이전 값으로 계속해서 변경되는 것을 보게 될 것이다. 또한, 여러분이 빈박스를 구동시킨 콘솔을 본다면 다음과 같은 메시지를 보게 될 것이다.

```
WARNING: Vetoed; reason is : from > to
```

보다 자세히 무슨 일들이 발생하는 지 살펴보자.

1. 여러분은 from 속성을 위한 편집 필드내에 값을 입력한다.
2. 빈박스는 RangeBean 클래스의 setFrom 메소드를 호출한다.
3. setFrom 메소드는 IntTextBean 클래스의 setValue 메소드를 호출한다.
4. setValue 메소드는 거부가능한 리스너들에게 이를 알린다. 우리의 경우 RangeBean이다.

5. RangeBean 클래스의 vetoableChange 메소드는 PropertyVetoException을 투구한다.
6. IntTextBean 클래스의 setValue 메소드는 예외상황을 투구하지 않는다. 그리고 값은 갱신되지 않는다.
7. 빈 박스는 예외상황을 감지하고 경고 메시지를 출력한다.
8. 빈 박스는 텍스트 필드내에 본래의 값으로 복귀하기 위해 getValue를 호출한다.

다음은 시도해 볼 다른 실험이다. 속성 시트내에 입력하는 것 대신에 from 텍스트 필드내에 직접 입력한다. 하위 경계에 상위 경계보다 큰 값을 입력하라. 그런 다음 탭을 누르거나 마우스를 다른 임의의 곳에 클릭하여 텍스트 필드를 내버려 둔다. IntTextBean은 사용자에게 거부 메시지를 출력하기 위해 다이얼로그를 팝업할 것이다.(그림 8-18)

그림 8-18 : 거부 메시지

이러한 에러는 다른 방법으로 제거할 수 있다. 텍스트 필드에서 포커스 이벤트를 추적하는 *포커스 리스너*를 설정한다. 텍스트 필드가 포커스를 얻으면 본래의 값을 저장한다. 포커스를 포기할 때 editComplete 메소드가 호출된다.

```
public IntTextBean(int defval, int size)
   {  super("" + defval, size);
      addFocusListener(new FocusListener()
         {  public void focusGained(FocusEvent event)
            {  if (!event.isTemporary())
               {  lastValue = getValue();
               }
            }
            public void focusLost(FocusEvent event)
            {  if (!event.isTemporary())
               {  editComplete();
               }
            }
         });
   }
```

editComplete 메소드는 setValue 메소드와 유사하다. 그러나 만일 속성 변경이 거부되면 본래의 값으로 복원하고 텟스트 필드의 포커스를 재설정한다.

```
public void editComplete()
  {  Integer oldValue = new Integer(lastValue);
     Integer newValue = new Integer(getValue());
      try
     {   fireVetoableChange("value", oldValue, newValue);
         // survived, therefore no veto
         firePropertyChange("value", oldValue, newValue);
      }
     catch(PropertyVetoException e)
     {  // someone didn't like it
        JOptionPane.showMessageDialog(this, "" + e,
            "Input Error", JOptionPane.WARNING_MESSAGE);
        setText("" + lastValue);
         requestFocus();
      }
  }
```

경고 : requestFocus **메소드는** JDK1.2**의 스윙 컴포넌트에 대해서는 작동하지 않는다-** http://developer.java.sun.com**에 있는 버그 퍼레이드의 버그**#4128659**를 참조하라**.

여러분의 빈에 제약 속성을 구성할 때마다 , 빈이 다중 리스너들을 갖고 이들 모두가 변경을 거부할 수 있다는 사실을 명심하라. 그러므로 바운드되지 않는 제약 속성은 그다지 합리적이지 않다. 리스너들은 요청한 속성 변경이 모든 리스너들에 의해서 실제 받아들여졌을 때 이를 감지할 필요가 있다. 이러한 이유 때문에, 우리는 모든 제약 속성들이 바운드되게 끔 하길 강력히 제안한다.

예제 8-3 : IntTextBean.java

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.io.*;
import javax.swing.*;
```

```
import javax.swing.text.*;

public class IntTextBean extends JTextField
    implements Serializable
{   public IntTextBean()
    {   this(0, 10);
    }

   public IntTextBean(int defval, int size)
    {   super("" + defval, size);
        addFocusListener(new FocusListener()
           {   public void focusGained(FocusEvent event)
               {   if (!event.isTemporary())
                   {   lastValue = getValue();
                    }
               }
               public void focusLost(FocusEvent event)
               {   if (!event.isTemporary())
                   {   editComplete();
                   }
               }
           });
    }

   public void editComplete()
   {   Integer oldValue = new Integer(lastValue);
       Integer newValue = new Integer(getValue());
        try
       {   fireVetoableChange("value", oldValue, newValue);
          // survived, therefore no veto
           firePropertyChange("value", oldValue, newValue);
        }
       catch(PropertyVetoException e)
       {   // someone didn't like it
          JOptionPane.showMessageDialog(this, "" + e,
              "Input Error", JOptionPane.WARNING_MESSAGE);
```

```
            setText("" + lastValue);
             requestFocus();
               // doesn't work in all JDK versions--see bug #4128659
          }
      }

    public int getValue()
     {   try
         {   return Integer.parseInt(getText());
         }
        catch (NumberFormatException exception)
         {   return 0;
         }
     }

    public void setValue(int v) throws PropertyVetoException
     {   Integer oldValue = new Integer(getValue());
        Integer newValue = new Integer(v);
         fireVetoableChange("value", oldValue, newValue);
        // survived, therefore no veto
         setText("" + v);
         firePropertyChange("value", oldValue, newValue);
     }

     protected Document createDefaultModel()
     {   return new IntTextDocument();
     }

     public Dimension getMinimumSize()
     {   return new Dimension(XMINSIZE, YMINSIZE);
     }

     private int lastValue;
    private static final int XMINSIZE = 50;
     private static final int YMINSIZE = 20;
}
```

```
class IntTextDocument extends PlainDocument
{  public void insertString(int offs, String str,
        AttributeSet a)
        throws BadLocationException
   {  if (str == null) return;
      String oldString = getText(0, getLength());
       String newString = oldString.substring(0, offs)
          + str + oldString.substring(offs);
        try
       {   Integer.parseInt(newString + "0");
          super.insertString(offs, str, a);
        }
       catch(NumberFormatException e)
        {
        }
    }
}
```

예제 8-4 : RangeBean.java

```
import java.awt.*;
import java.beans.*;
import java.io.*;
import javax.swing.*;

public class RangeBean extends JPanel
   implements VetoableChangeListener, Serializable
{   public RangeBean()
    {   add(new JLabel("From"));
        add(from);
       add(new JLabel("To"));
        add(to);

        from.addVetoableChangeListener(this);
        to.addVetoableChangeListener(this);
```

```
    }

   public void vetoableChange(PropertyChangeEvent event)
        throws PropertyVetoException
   {   int v = ((Integer)event.getNewValue()).intValue();
     if (event.getSource() == from && v > to.getValue())
          throw new PropertyVetoException("from > to", event);
     if (event.getSource() == to && v < from.getValue())
          throw new PropertyVetoException("to < from", event);
    }

   public int getFrom() { return from.getValue(); }
   public int getTo() { return to.getValue(); }

   public void setFrom(int v) throws PropertyVetoException
    {   from.setValue(v);
    }

   public void setTo(int v) throws PropertyVetoException
    {   to.setValue(v);
    }

   private IntTextBean from = new IntTextBean();
   private IntTextBean to = new IntTextBean();
}
```

java.bean.VetoableChangeListener

- void vetoableChange(PropertyChangeEvent event)
  속성이 변경되려고 할 때 호출된다. 만일 해당 변경이 받아들여질 수 없다면 PropertyVetoException**을 투구할 것이다.**
  전달인자: event **속성 변경을 묘사하는 이벤트 객체**

java.bean.VetoableChangeSupport

- VetoableChangeSupport(Object sourceBean)
  VetoableChangeSupport**객체를 구성한다.**

전달인자: sourceBean 변경가능한 변화의 소스 빈(보통 this이다.)

- void addVetoableChangeListener(VetoableChangeListener listener)
  제약 속성에 대해 관심있는 리스너를 등록한다.
  전달인자: listener 제약 속성을 거부할 기회를 갖길 원하는 빈

- void removeVetoableChangeListener(VetoableChangeListener listener)
  제약 속성에 대해 관심있는 리스너를 제거한다.
  전달인자: listener 리스너 목록에서 제거해야 할 빈

- void fireVetoableChange(String propertyName, Object oldValue, Object newValue)
  속성 값을 변경하기에 앞서 등록된 리스너들에게 PropertyChangeEvent를 보낸다.
  전달인자: propertyName 속성 이름
  oldValue 현재 값
  newValue 제안된 새로운 값

javax.swing.JComponent

- void addVetoableChangeListener(PropertyChangeListener listener)
  해당 컴포넌트의 모든 제약 속성들에 대해 관심있는 리스너를 등록한다.
  전달인자: listener 바운드 속성에서 변경을 알길 원하는 객체

- void removeVetoableChangeListener(PropertyChangeListener listener)
  해당 컴포넌트의 제약 속성들에 대해 이전에 등록된 관심있는 리스너들을 제거한다.
  전달인자: listener 리스너 목록에서 제거될 객체

- void fireVetoableChange(String propertyName, Object oldValue, Object newValue)
  등록된 리스너들에게 PropertyChangeEvent를 보낸다.
  전달인자: propertyName 속성 이름
  oldValue 이전 값
  newValue 새로운 값

java.beans.PropertyVetoException

- PropertyVetoException(String reason , PropertyChangeEvent event)
  새로운 PropertyVetoException을 생성한다.
  전달인자: reason 거부하는 이유를 설명하는 문자열
  event 거부하길 원하는 제약 속성에 대한

PropertyChangeEvent

- PropertyChangeEvent getPropertyChangeEvent()
  예외상황을 생성하기 위해 사용하는 PropertyChangeEvent**을 반환한다.**

사용자 빈 이벤트 추가하기

여러분이 빈에 바운드나 제약 속성을 추가할 때, 해당 속성의 값이 변할 때마다 빈에서 이벤트들을 발생하게 할 수 있다. 그러나 빈에서 발생시킬 수 있는 다른 이벤트가 있다. 예를 들면,

- **프로그램 사용자가 빈에 있는 컨트롤을 클릭했을 때**
- **새로운 정보가 이용가능해 질 때**
- **또는, 일정 시간이 경과됐을 때**

PropertyChangeEvent **이벤트와 달리, 이들 이벤트들은 커스텀 클래스에 속하고 커스텀 리스너에 의해서 감지될 필요가 있다.**

다음은 커스텀 이벤트들을 생성하는 빈을 작성하는 방법이다. (자바 플랫폼에서 이벤트 핸들링에 대한 보다 자세한 내용은 1**권의** 8**장을 참조하라.) 첫번째 두 과정을 정교하게 따라야 한다. 그렇지 않으면 내성(인트로스펙트,** Introspect**) 메커니즘이 여러분이 정의하려고 노력하는 커스텀 이벤트를 인식하지 못할 것이다.**

1. EventObject**를 확장한** CustomEvent **클래스를 작성한다. (빌더가 찾을 수 있는 명명 패턴을 사용하기 위해 이벤트 이름은** Event**내에서 끝나야 한다.)**
2. **단일 알림**(notification) **메소드를 갖는** CustomListner **인터페이스를 작성한다. 이 알림 메소드는 임의의 이름을 갖을 수 있다. 그러나** CustomEvent **형의 단일 전달인자와** void **반환 형을 가져야 한다.**
3. **빈내에 다음 두개의 메소드를 제공해야 한다.**
   public void addCustomListener(CustomListener e)
   public void removeCustomListener(CustomListener e)

경고 : 만일 여러분의 이벤트 클래스가 EventObject**를 확장하지 않았다면, 여러분의 코드는 문제없이 컴파일될 것이다. 왜냐하면** EventObject **클래스의 메소드들중 실제로 필요한 것은 아무것도 없기 때문이다. 그리고 이벤트 객체를** EventObject **클래스에 형변환을 시도하는 다른 메소드도 없다. 그러나 여러분의 빈은 실패할 것이다 – 내성 메커니즘이 이벤트들을 인식하지 못할 것이다.**

커스텀 이벤트들을 추가하고 제거하고 전달하는 데 필요한 메소드를 구현하기 위해, 여러분은 자동적으로 이벤트 리스너들을 관리하는 편리한 클래스들에 더 이상 의존할 수 없다. 대신에, 여러분은 모든 이벤트 리스너를 모으고 관리할 필요가 있다. 가령, Vector에서 이벤트를 전달할 때 여러분은 또한 모든 수집된 리스너들에 대한 알림(notification) 메소드를 호출할 필요가 있다. 이 호출은 동기화 문제를 초래할 수 있다. – 한 쓰레드가 리스너에 추가하거나 제거하려 하는 동시에 다른 쓰레드가 이벤트를 전달을 처리할 때 발생할 수 있다. 다음 코드는 필요한 동기화를 제공하는 방법을 보여준다.

```
public synchronized void addCustomListener( CustomListener l)
{       listeners.addElement(l);
}

public synchronized void removeCustomListener( CustomListener l)
{       listeners.removeElement(l);
}

public void fireCustomEvent( CustomEvent event)
{       Vector currentListeners = null;
        synchronized(this)
        {       currentListeners = (Vector)listeners.clone();
        }
        for(int i=0; i < currentListeners.size(); i++)
        {       CustomListener listener = (CustomListener) currentListeners.elementAt(i);
                listener.notifyMethod(event);
        }
}
…
private Vector listeners = new Vector();
```

동기화를 위한 코드는 다소 애매하다. 왜 synchronized 메소드내에 fireCustomEventfmf 만들지 않는 이유는 무엇일까? 이러면, 우리는 발생할 이벤트로서 추가하거나 제거되는 리스너들은 없다는 것을 알고 있어야 한다. 그러나 잠재적으로 데드락으로 이끌수 있다. 왜내하면 몇몇의 경우 notifyMethod는 addCustomerListener나 removeCustomerListener를 호출할 수 있기 때문이다. 별도의 쓰레드내에서 호출되지 않기 때문에 해당 호출이 데드락을 발생시킬 수 있다. 그러므로 우리는 먼저 벡터를 복제한다. 오직 복제를 수행하는 코드 블록만이 동기화되어

있다. 물론 , 현재 오직 전달 과정의 시작에 등록된 리스너들만이 알려진다. 그리고 만일 한 리스너가 전달 과정이 완료되기 전에 제거된다면 해당 리스너는 여전히 호출된다. 이것은 반대로 리스너 그 자체가 이벤트 소스로부터 제거될 때, 즉시 이벤트 전송이 중단될 것이라는 것을 절대적으로 보장하지 못한다는 것을 의미한다. 반대로, 만일 리스너가 이벤트의 전달 과정 중간에 추가된다면 새롭게 추가된 리스너는 복제된 벡터의 일부가 아니다. 그리고 같은 이벤트를 전파(notify)하지 않는다.

팁 : 만일 여러분이 직접 리스너 목록을 구현하길 원하지 않는다면, javax.swing.EventListenerList를 사용할 수 있다. 보다 자세한 것은 6장을 참조하라.

이제, 이 기술을 TimerBean을 구현하는 제 적용해 보자. 이 빈은 TimerEvent 객체를 해당 리스너에게 보낼 것이다. (이것은 2장의 Timer 클래스의 변형이다.) 타이머 이벤트는 일정한 간격을 생성하고 밀리 초로 측정하고 true로 설정된 running 속성을 제공한다. 예제 8-5에서부터 8-7는 다음 코드를 보여준다.

- TimerEvent 클래스 , 해당 빈에 의해서 생성된 커스텀 이벤트
- TimeElapsed 라 불리는 통지 메소드를 갖는 TimerListener 클래스
- addTimerListener와 removeTimerListener 메소드를 갖는 TimerBean

다음은 이 빈을 테스트하는 방법이다. 타이머 빈을 빈박스에 떨어뜨린다. (이것은 paint 메소드를 갖지 않는 보이지 않는 빈이다. – 이것은 단지 빈박스에 의해서 “TimerBean” 문자열을 출력한다.) 그런 다음, 빈박스에 EventMonitor 빈을 떨어트린다. TimerBean을 선택하고 빈박스 메뉴로부터 Edit->Events를 선택한다. TimeElapsed 자식 메뉴를 갖는 timer 하위 메뉴를 주목하라. 이들 메뉴들은 커스텀 이벤트를 제대로 식별하는 내성 메소드를 보여준다. TimeElapsed 메소드를 선택하라. 그러면 적색 선이 나타난다. 이 선을 EventMonitor에 연결한다. 이벤트 통지가 발생할 때 다이얼로그가 여러분에게 어느 메소드를 호출할 것인지 묻는다. 오직 이용가능한 메소드인 initiateEventSourceMonitoring을 선택한다. 이제, 다시 TimerBean을 선택한다. 속성 시트에서 running 속성을 true로 설정하고 EventMonitor빈 내의 텍스트 영역을 감시한다. 매 초마다 통지 메시지가 출력된다.(그림 8-19 참조)

그림 8-19 : 커스텀 이벤트의 빈박스 모니터

다음은 TimerBean의 완전한 코드이다.

예제 8-5 : TimerBean.java

```
import java.awt.*;
import java.util.*;
import java.io.*;

public class TimerBean implements Runnable, Serializable
{  public int getInterval() { return interval; }
   public void setInterval(int i) { interval = i; }

   public boolean isRunning() { return runner != null; }
   public void setRunning(boolean b)
   {  if (b && runner == null)
      {   runner = new Thread(this);
          runner.start();
      }
     else if (!b && runner != null)
      {   runner.interrupt();
         runner = null;
      }
   }

   public synchronized void addTimerListener
       (TimerListener l)
   {   timerListeners.addElement(l);
   }

   public synchronized void removeTimerListener
       (TimerListener l)
   {   timerListeners.removeElement(l);
   }

   public void fireTimerEvent(TimerEvent evt)
   {  Vector currentListeners = null;
       synchronized(this)
      {   currentListeners = (Vector)timerListeners.clone();
       }
```

```
      for (int i = 0; i < currentListeners.size(); i++)
        {   TimerListener listener
                = (TimerListener)currentListeners.elementAt(i);
            listener.timeElapsed(evt);
        }
    }

   public void run()
   {  if (interval <= 0) return;
       try
       {   while (!Thread.interrupted())
           {   Thread.sleep(interval);
               fireTimerEvent(new TimerEvent(this));
           }
       }
       catch(InterruptedException e)
       {
       }
    }

   private int interval = 1000;
   private Vector timerListeners = new Vector();
    private Thread runner;
}
```

예제 8-6 : TimerListner.java

```
import java.util.*;

public interface TimerListener extends EventListener
{  public void timeElapsed(TimerEvent evt);
}
```

예제 8-7 : TimerEvent.java

```
import java.util.*;
```

```
public class TimerEvent extends EventObject
{   public TimerEvent(Object source)
    {   super(source);
        now = new Date();
    }

  public Date getDate() { return now; }

   private Date now;
}
```

속성 편집기

만일 여러분이 정수나 문자열 속성을 빈에 추가한다면 , 해당 속성은 자동적으로 빈의 속성 시트를 출력한다. 그러나 만일 텍스트 필드내에서 쉽게 편집할 수 없는 값을 갖은 속성을 추가한다면 무슨 일이 발생할 까? 가령, 날짜나 Color같은? 그러면 여러분은 사용자가 속성 값을 명시할 수 있게 해주는 별도의 컴포넌트를 제공할 필요가 있다. 이와 같은 컴포넌트를 속성 편집기라 부른다. 예를 들면, 날짜 객체를 위한 속성 편집기는 사용자가 달을 넘기고 날짜를 선택하게 해주는 달력일 수 있다. Color객체의 속성 편집기는 사용자에게 색의 적,녹, 청 컴포넌트를 선택하게 해준다.

실제, 빈박스는 이미 색상을 위한 속성 편집기를 가지고 있다 – 여러분은 그림 8-4에서 이미 보았다. 물론 String(텍스트 필드)와 Boolean(true와 false 값을 갖는 선택 목록)같은 기본 형들에 데힌 속성 편집기가 존재한다. 이들 속성 편집기는 *속성 편집 관리자*에 등록되어 있다. 여러분은 속성 편집기를 PropertyEditiorManager 클래스의 registerEditor 정적 메소드를 갖는 관리자에 추가해야 한다. 여러분은 에디터 클래스와 에디터에 적용하는 클래스들을 제공해야 한다.

```
PropertyEditorManager.registerEditor(Date.class, CalendarSelector.class);
```

여러분은 빌더 환경에서 주어진 형에 대해 속성 편집기가 존재하는 지를 검사하기 위해 PropertyEditorManager 클래스의 findEditor 메소드를 사용할 수 있다. 해당 메소드는 다음을 수행한다:

1. 먼저, 어느 속성 편집자들이 이미 등록되었는지 살펴본다.
   (이들은 빌더 툴에서 제공되는 편집기들이다. 그리고 여러분은 registerEditor를 호출하여 제공한다.)
2. 그런 다음, Editor 단어와 형 이름으로 구성되는 이름을 갖는 클래스를 찾는다.
3. 만일, 룩업이 성공하지 못하면 findEditor는 null을 반환한다.

가령, CalendarSelector 클래스가 Date 객체를 위해 등록되었다면 이 클래스는 Date 속성을 편집하기 위해 사용되어 진다.
또한 빈 박스는 속성 시트내에서 출력되는 각 타입을 위한 편집기를 위치시키기 위해 findEditor 메소드를 사용한다. 그러나 일반(Generic) 편집기를 찾기 전에 여러분의 빈의 *빈 정보*내에서 특정 편집기를 여러분이 요청했는지 검사한다. 빈 정보는 여러분의 빈에 대한 갖가지 정보의 집합이다. 가령, 만일 여러분이 int나 String 형의 속성을 갖고 이들의 합법적인 값들은 여러가지 방법으로 제약받는다면 여러분은 빈 박스에서 제공하는 일반 목적의 속성 편집기를 사용하길 원치 않을 것이다. 대신에 여러분은 특정 속성을 위한 특정 편집기를 제공할 수 있다. 이를 위해 빈 정보내에 특정 편집기에 이름을 부여한다.

특정 속성 편집기를 제공하는 과정은 다음의 과정을 포함하고 있다. 먼저, 여러분은 여러분의 빈에 대한 빈 정보 클래스를 생성한다. 클래스의 이름은 BeanInfo로 시작하는 빈의 이름이 돼야 한다. 해당 클래스는 8개의 메소드를 갖고 있는 BeanInfo 인터페이스를 구현해야 한다. 이대신 SimpleBeanInfo 클래스를 확장하는 것이 가장 단순하다. 이 편리한 클래스는 8개의 메소드에 대해 아무 것도 구현하지 않는다. 가령,

```
// bean info class for chartBean
class ChartBeanBeanInfo extends SimpleBeanInfo
{           ….
}
```

특정 빈을 위한 특정 편집기를 요청하기 위해, 여러분은 getPropertyDescriptors 메소드를 오버라이드한다. 해당 메소드는 PropertyDescriptor 객체의 배열을 반환한다. 비록 여러분이 기본 편집기만을 원한다 할 지라도, 여러분은 속성 편집기상에 출력되는 각 속성마다 하나의 객체를 생성한다.

여러분은 속성의 이름과 이를 포함하고 있는 빈의 클래스를 제공함으로써 PropertyDescriptor를 구성한다.

```
PropertyDescriptor descriptor = new PropertyDescriptor(“titlePosition” , ChartBean.class);
```

속성에 대한 특정 편집기를 요청하려면, 여러분은 PropertyDescriptor클래스의

setPropertyEditorClass를 호출한다.

descriptor. setPropertyEditorClass(TitlePositionEditor.class);

다음, 여러분의 빈의 속성들을 설명하는 배열을 구성한다. 가령, 이 절에서 설명하는 챠트 빈은 4개의 속성을 가지고 있다:

- String 속성 , title
- int 속성 , titlePosition
- double[] 속성, values
- boolean 속성 , inverse

그림8-20은 챠트 빈을 보여준다. 여러분은 상단에 제목을 볼 수 있다. 위치는 왼쪽, 중간, 또는 오른쪽에 위치할 수 있다. values 속성은 그래프의 값을 명시한다. 만일 inverse 속성이 true이면 배경은 컬러화되고 채트의 바들은 흰색이 된다. 예제 8-8은 챠트 빈의 코드이다. 빈은 1권의 10장에 있는 챠트 애플릿의 간단한 변형이다.

그림 8-20 : 챠트 빈

예제 8-9의 코드는 이러한 속성들에 대한 속성 편집기를 명시하는 ChartBeanBeanInfo 클래스를 보여준다. 내용은 다음과 같다:

1. static 블록에서, DoubleArrayEditor는 임의의 double[] 배열을 위한 편집기로 등록된다.
2. getPropertyDescriptors 메소드는 각 속성에 대한 설명을 반환한다. title과 values 속성은 기본 편집기들과 함께 사용된다. 즉, 빈박스에서 제공하는 문자열 편집기와 static 블록에 등록되어 있는 DoubleArrayEditor이다.
3. titilePosition과 inverse 속성들은 각기 TitlePositionEditor와 InverseEditor 형의 특별한 편집기들을 사용한다.

그림 8-21은 결과 속성 시트를 보여준다. 여러분은 다음 절에서 이러한 종류의 편집기들을 구현하는 방법을 보게 될 것이다.

그림 8-21 : 챠트 빈의 속성 시트

예제 8-8 : ChartBean.java

```
import java.awt.*;
import java.util.*;
```

```
import java.beans.*;
import java.io.*;
import javax.swing.*;

public class ChartBean extends JPanel
    implements Serializable
{   public void paint(Graphics g)
    {   if (values == null || values.length == 0) return;
         int i;
        double minValue = 0;
        double maxValue = 0;
          for (i = 0; i < values.length; i++)
        {   if (minValue > getValues(i)) minValue = getValues(i);
            if (maxValue < getValues(i)) maxValue = getValues(i);
          }
        if (maxValue == minValue) return;

        Dimension d = getSize();
         int clientWidth = d.width;
        int clientHeight = d.height;
       int barWidth = clientWidth / values.length;

        g.setColor(inverse ? color : Color.white);
        g.fillRect(0, 0, clientWidth, clientHeight);
         g.setColor(Color.black);

        Font titleFont = new Font("SansSerif", Font.BOLD, 20);
        FontMetrics titleFontMetrics
            = g.getFontMetrics(titleFont);

        int titleWidth = titleFontMetrics.stringWidth(title);
        int y = titleFontMetrics.getAscent();
         int x;
        if (titlePosition == LEFT)
            x = 0;
        else if (titlePosition == CENTER)
```

```
            x = (clientWidth - titleWidth) / 2;
        else
            x = clientWidth - titleWidth;

        g.setFont(titleFont);
        g.drawString(title, x, y);

        int top = titleFontMetrics.getHeight();
        double scale = (clientHeight - top)
            / (maxValue - minValue);
        y = clientHeight;

      for (i = 0; i < values.length; i++)
      {   int x1 = i * barWidth + 1;
          int y1 = top;
          int height = (int)(getValues(i) * scale);
          if (getValues(i) >= 0)
              y1 += (int)((maxValue - getValues(i)) * scale);
            else
          {   y1 += (int)(maxValue * scale);
              height = -height;
            }

          g.setColor(inverse ? Color.white : color);
          g.fillRect(x1, y1, barWidth - 2, height);
            g.setColor(Color.black);
          g.drawRect(x1, y1, barWidth - 2, height);
        }
    }
  public void setTitle(String t) { title = t; }
  public String getTitle() { return title; }


   public  double[] getValues() { return values; }

  public void setValues(double[] v) { values = v; }
```

```
public double getValues(int i)
{  if (0 <= i && i < values.length) return values[i];
     return 0;
 }

public void setValues(int i, double value)
 {   if (0 <= i && i < values.length) values[i] = value;
 }

 public boolean isInverse()
 {   return inverse;
 }

public void setTitlePosition(int p) { titlePosition = p; }

 public int getTitlePosition()
 {   return titlePosition;
 }

  public void setInverse(boolean b) { inverse = b; }

 public Dimension getMinimumSize()
 {   return new Dimension(MINSIZE, MINSIZE);
 }

public void setGraphColor(Color c) { color = c; }
public Color getGraphColor() { return color; }

 private static final int LEFT = 0;
private static final int CENTER = 1;
private static final int RIGHT = 2;

private static final int MINSIZE = 50;
private double[] values = { 1, 2, 3 };
private String title = "Title";
```

```
    private int titlePosition = CENTER;
    private boolean inverse;
    private Color color = Color.red;
}
```

예제 8-9 : ChartBeanBeanInfo.java

```
import java.beans.*;

public class ChartBeanBeanInfo extends SimpleBeanInfo
{   public PropertyDescriptor[] getPropertyDescriptors()
    {   try
        {   PropertyDescriptor titlePositionDescriptor
              = new PropertyDescriptor("titlePosition",
                    ChartBean.class);
            titlePositionDescriptor.setPropertyEditorClass
                (TitlePositionEditor.class);
            PropertyDescriptor inverseDescriptor
              = new PropertyDescriptor("inverse",
                    ChartBean.class);
            inverseDescriptor.setPropertyEditorClass
                (InverseEditor.class);

            return new PropertyDescriptor[]
            {   new PropertyDescriptor("title",
                    ChartBean.class),
                titlePositionDescriptor,
                new PropertyDescriptor("values",
                    ChartBean.class),
                new PropertyDescriptor("graphColor",
                    ChartBean.class),
                inverseDescriptor
            };
        }
```

```
        catch(IntrospectionException e)
        {   System.out.println("Error: " + e);
            return null;
        }
    }

    static
    {   PropertyEditorManager.registerEditor(double[].class,
            DoubleArrayEditor.class);
    }
}
```

java.beans.PropertyEditorManager

- static PropertyEditor findEditor(Class targetType)
  주어진 형에 대한 속성 편집기를 반환하거나 등록된 것이 없다면 null**을 반환한다**.
  전달인자: targetType Class.Color**처럼 편집할 형에 대한** Class **객체**

- static void registerEditor(Class targetType, Class editorClass)
  주어진 형의 값들을 편집할 편집기 클래스를 등록한다.
  전달인자: targetType **편집할 형에 대한** Class **객체**
  editorClass **편집기 클래스에 대한** Class **객체(현재 편집기가 등록되지 않았으면** null**이다**.)

java.beans.PropertyDescriptor

- PropertyDescriptor(String name, Class beanClass)
  PropertyDescriptor **객체를 구성한다**.
  전달인자: name **속성의 이름**
  beanClass **속성이 속해있는 빈의 클래스**

- void setPropertyEditorClass(Class editorClass)
  해당 속성에서 사용되어 지는 속성 편집기의 클래스를 설정한다.

java.beans.BeanInfo

- PropertyDescriptor[] getPropertyDescriptors()
  해당 빈에 대한 속성 시트에서 출력되어지는 각 속성에 대한 설명을 반환한다.

속성 편집기 작성

속성 편집기를 작성하는 방법을 보여주기 전에, 우리는 각 속성 편집기가 하나의 특정한 형의 값으로 작업을 하는 과정이 아주 복잡하지 않다는 것을 지적하고 싶다. 예를 들면, (Color형의 객체를 편집하는) 컬러 속성 편집기는 슬라이더나 팔레트(또는 두개 모두)를 사용하여 사용자가 보다 적합한 방법으로 컬러를 편집하게 해준다.

다음, 여러분이 작성한 임의의 속성 편집기는 12개의 메소드를 갖고 있는 PropertyEditor 인터페이스를 구현해야 한다. BeanInfo 인터페이스에서 처럼, 여러분은 직접 이 작업을 하길 원치 않을 것이다. 대신, 표준 라이브러리에서 제공하는 PropertyEditorSupport 클래스를 확장하는 것이 보다 편리하다. 이러한 지원 클래스는 프로퍼티 변경 리스너들을 추가하고 제거하는 메소드와 PropertyEditor 인터페이스의 모든 다른 메소드들의 기본 버전을 갖고 있다. 예를 들면, 우리의 챠트 빈에서 챠트의 제목 위치를 편집하기 위한 우리의 편집기는 다음과 같다:

```
// property editor class for title position
class TitlePositionEditor extends PropertyEditorSupport
{           …
}
```

만일, 속성 편집기 클래스가 하나의 생성자를 갖는 다면 전달인자없는 기본 생성잘ㄹ 제공해야 한다는 것을 기억하라.

마지막으로, 실제 속성 편집기를 작성하는 방법으로 들어가기 전에, 우리는 빈이 아니라 빌더의 제어하에 있는 편집기를 지적하고 싶다. 빌더는 속성의 현재 값을 출력하기 위해 다음의 과정을 밟는다.

1. **빈의 각 속성에 대한 속성 편집기들을 초기화한다.**
2. ***빈*에게 속성의 현재 값을 알려달라고 요청한다.**
3. **그런 다음 해당 값을 출력하라고 *속성 편집기*에게 요청한다.**

속성 편집기는 실제 값을 출력하기 위해 텍스트나 그래픽 기반의 메소드를 사용할 수 있다. 다음 절에서 이러한 메소드들에 대해 논의한다.

단순 속성 편집기

단순 속성 편집기는 텍스트 문자열로 작업을 한다. 여러분은 setAsText와 getAsText 메소드를 오버라이드한다. 가령, 우리의 챠트 빈은 출력될 제목의 위치를 설정하게 해주는 속성을 가

지고 있다 : 왼쪽, 중앙, 오른쪽. 이러한 선택들은 정수 상수로서 구현된다.

```
private static final int LEFT = 0;
private static final int CENTER = 1;
private static final int RIGHT = 2;
```

물론, 우리는 이들을 텍스트 필드내에 숫자 0,1,2로 표기하길 원하지 않는다. 따라서 우리는 문자열로 값을 반환하는 getAsText 메소드를 갖는 속성 편집기를 정의한다. PropertyEditor의 getValue 메소드 호출은 속성의 값을 발견한다. 이 메소드는 기본(Generic) 메소드이기 때문에 반환되는 값은 Object 형이다. 만일 속성 타입이 기본 타입이면, 우리는 포장(Wrapper) 객체를 반환할 필요가 있다. 우리의 경우, 속성 타입이 int 이기 때문에 getValue호출은 Integer 를 반환한다.

```
class TitlePositionEditor extends PropertyEditorSupport
{       public String getAsText()
        {       int i = ((Integer)getValue()).intValue();
                if( 0 <= i && i < options.length) return options[i];
                return " ";
        }
        ….
        private String[] options = { "left" , "Center", "Right" };
}
```

이제, 텍스트 필드는 이들 필드중 하나를 출력한다. 사용자가 텍스트 필드를 편집할 때, setValue 메소드를 속성의 값을 갱신하는 setAsText 메소드에 대한 호출을 트리거한다. 이 또한 Object 타입의 전달인자를 갖는 일반 메소드이다. 숫자 타입의 값을 설정하기 위해, 우리는 포장 객체를 전달할 필요가 있다.

```
public void setAsText(String s)
{       for(int i =0; i < options.length; i++)
        {       if(options[i].equals(s))
                {       setValue(new Integer(i) );
                        return;
                }
        }
}
```

실제로, 이 속성 편집기는 보기 민망한 사용자 인터페이스를 만들려고 하지 않는한 titlePosition 속성에 대해 좋은 선택은 아니다. 사용자는 어떤 선택이 합법적인 것인지 알지 못한다. 모든 가능한 설정을 출력하는 편이 보다 낳다. (그림 8-22참조) PropertyEditorSupport 클래스는 속성 편집기에서 선택항목을 출력하는 단순한 메소드를 제공한다. 우리는 문자열의 배열을 반환하는 getTags 메소드를 간단히 작성한다.

```
public String[] getTags() { return options; }
```

기본 getTags 메소드는 널(null)을 반환한다. 널이 아닌 값을 반환하는 것에 의해서, 텍스트 필드대신 선택 필드를 나타낼 수 있다.

우리는 여전히 getAsText와 setAsText 메소드를 제공할 필요가 있다. getTags 메소드는 Choice 필드내에 출력되어지는 값들을 명시한다. GetAsText/setAsText 메소드는 속성의 자료형과 문자열사이를 변환한다(속성의 자료형은 문자열이나 정수 또는 완전히 다른 형일 수 있다).

그림 8-22 : 작동중인 PositionEditor

예제 8-10은 속성 편집기의 완전한 코드이다.(실제 빈에 대한 코드는 예제 8-8을 참조하라.)

예제 8-10 : TitlePositionEditor.java

```
import java.beans.*;

public class TitlePositionEditor
    extends PropertyEditorSupport
{   public String getAsText()
    {   int value = ((Integer)getValue()).intValue();
        return options[value];
     }

    public void setAsText(String s)
    {   for (int i = 0; i < options.length; i++)
        {   if (options[i].equals(s))
            {   setValue(new Integer(i));
                return;
             }
```

```
        }
    }

  public String[] getTags() { return options; }

  private String[] options = { "Left", "Center", "Right" };
}
```

java.beans.PropertyEditorSupport

- Object getValue()
  속성의 현재 값을 반환한다. 기본 형들은 객체 포장자들에 포장되어 있다.
- void setValue(Object newValue)
  새로운 값으로 속성을 설정한다. 기본 형들은 객체 포장자들에 포장되어 있다.
  전달인자: newValue **객체의 새로운 값; 속성이 갖을 수 있는 새롭게 생성된 객체일 수 있다.**
- String getAsText()
  속성의 현재 값의 문자열 표현을 반환하기 위해 이 메소드를 오버라이드한다. 기본은 속성이 문자열로 표현될 수 없다는 것을 나타내는 null**을 반환한다.**
- void setAsText(String text)
  텍스트를 파싱하여 얻어지는 새로운 값으로 속성을 설정하기 위해 이 메소드를 오버라이드한다. 만일 텍스트가 합법적인 값을 표현하지 않거나 속성이 문자열로 표현될 수 없다면 IllegalArgumentException**을 발생한다.**
- String[] getTags()
  선택 박스내에서 속성들을 보여지게 하기 위해 속성들의 값들의 모든 가능한 문자열 배열을 반환한다. 기본은 한정된 문자열 값들이 아니라는 것을 나타내는 null**을 반환한다.**

GUI기반의 속성 편집기

보다 복잡한 속성 타입들은 텍스트로서 편집될 수 없다. 대신, 이들은 두가지 방법으로 표현된다. 속성 시트는 속성 편집기에서 현재 값의 그래픽 표현을 그릴 수 있는 작은 공간(텍스트 박스나 선택 필드를 유지할 수도 있음)을 포함하고 있다. 사용자가 이 공간을 클릭할 때, 사용자 편집기 다이얼로그 박스가 팝업된다(그림 8-23**참조). 다이얼로그 박스는 속성 편집기에서 제공하는 속성 값들을 편집하는 컴포넌트들과 빈박스에서 제공하는** Done **제목의 버튼을 포함하고 있다.**

그림 8-23 : 사용자 편집기 다이얼로그

GUI기반의 속성 편집기를 구성하기 위해:

1. 빌더 도구에게 문자열을 사용하기 않고 값을 그릴 거라고 말한다.
2. GUI상에 입력한 값을 "Paint" 한다.
3. 빌더 도구에게 GUI 기반의 속성 편집기를 사용할 것이라고 말한다.
4. GUI를 구성한다.
5. 사용자가 입력한 값의 정당성을 검사하는 코드를 작성한다.

첫번째 단계를 위해, 여러분은 PropertyEditor 인터페이스의 getAsText 메소드를 null을 반환하고 isPaintable 메소드가 true를 반환하게 끔 오버라이드 한다.

```
public String getAsText()
{       return null;
}
public boolean isPaintable()
{       return true;
}
```

그럼 다음, 여러분은 paintValue 과정을 구현한다. Graphics 핸들과 그릴 수 있는 사각형의 좌표를 받는다. 이 사각현은 기본적으로 작기 때문에 여러분은 아주 복잡한 표현을 할 수 없다는 것을 기억하라. Inverse 속성의 그래픽 표현을 위해, 우리는 검은 배경에 흰색 글자로 "Inverse" 문자열을 그리거나 흰색 배경에 검은 글자로 "Normal" 문자열을 그린다(그림 8-21 참조).

```
public void paintValue(Graphics g, Rectangle box)
  {  boolean isInverse = ((Boolean)getValue()).booleanValue();
     String s = isInverse ? "Inverse" : "Normal";
     g.setColor(isInverse ? Color.black : Color.white);
     g.fillRect(box.x, box.y, box.width, box.height);
     g.setColor(isInverse ? Color.white : Color.black);
     FontMetrics fm = g.getFontMetrics();
     int w = fm.stringWidth(s);
     int x = box.x;
     if (w < box.width) x += (box.width - w) / 2;
     int y = box.y + (box.height - fm.getHeight()) / 2
```

```
        + fm.getAscent();
    g.drawString(s, x, y);
}
```

물론, 이러한 그래픽 표현은 편집이 불가능하다. 사용자는 사용자 편집기를 팝업하기 이해 이 영역을 클릭해야 한다.

여러분은 PropertyEditor 인터페이스의 supportsCustomEditor가 true를 반환하도록 오버라이드한 사용자 편집기를 가져야 한다.

```
public boolean supportCustomEditor()
{       return true;
}
```

이제, 여러분은 사용자 편집기를 유지할 컴포넌트를 구성하는 코드를 작성한다. 여러분은 모든 속성을 위해 별도의 사용자 편집기 클래스를 구성할 필요가 있을 것이다. 예를 들면, InverseEditorPanel 클래스내에 InverseEditor 클래스와 관련하여 일반 모드와 반전 모드간의 선택이 가능한 두개의 레디오 버튼을 갖는 GUI를 묘사한다. 해당 코드는 아주 확연하다. 그러나, GUI 액션들은 속성 값들을 갱신해야 한다. 우리는 이를 다음과 같이 수행한다:

1. 속성 편집기 객체에 대한 참조를 받고 이를 editor변수에 저장하는 사용자 편집기 생성자를 갖는다.
2. 속성 값을 읽기위해, 우리는 다음을 호출하는 사용자 편집기를 갖는다.
   editor.getValue().
3. 객체 값을 설정하기 위해, 우리는 다음을 호출하는 사용자 편집기를 갖는다.
   editor.firePropertyChange() 이어서
   editor.setValue(newValue)

다음, PropertyEditor 인터페이스의 getCustomEditor 메소드를 구성하고 사용자 편집기 클래스의 객체를 반환한다.

```
public Component getCustomEditor()
{       return new InverseEditorPanel(this);
}
```

예제 8-11은 속성 시트내에 현재 설정을 보여주는 InverseEditor를 위한 전체 코드를 보여준

다. 예제 8-12는 팝업 편집기 패널을 구현한 코드이다.

챠트 빈 클래스를 위해 우리가 구성하는 다른 사용자 편집기는 여러분에게 double[] 배열을 편집하게 해준다. 빈박스가 배열 속성들을 편집할 수 없다는 것을 상기하라. 우리는 이러한 명백한 한계를 극복하기 위해 이러한 사용자 편집기를 개발하였다. 이러한 사용자 편집기는 약간만 포함되어 있다. 그림8-24는 작동중인 사용자 편집기를 보여준다. 모든 배열 값들은 자신들의 배열 색인으로 시작하며 리스트 박스내에서 보여진다. 배열 값을 클릭하여 위에 있는 텍스트 필드에 위치시키고, 이를 편집할 수 있다. 또한 여러분은 배열의 크기를 재조정할 수 있다. GUI를 구현한 DoubleArrayPanel 클래스의 코드는 예제 8-14에 있다.

속성 편집기 클래스의 코드(예제 8-13에 보여지는)는 paintValue 메소드에서 첫번째 몇몇 배열 값들로 구성되는 문자열을 그린다는 점을 제외하고 거의 InverseEditor의 코드와 거의 동일하다. 물론, 우리는 getCustomEditor 메소드에서 다른 사용자 편집기를 반환한다. 이러한 예제들은 차트 빈에 대한 완전한 코드이다.

노트 : 불운하게도, 우리는 배열 값들을 그려야만 했다. 이것은 geyAsText 메소드로 문자열을 반환하는 것보다 편리한 방법이다. 그러나 빈 박스는 해당 텍스트를 편집할 수 있다고 가정한다. 그리고 만일 여러분이 이를 클릭한다면 비록 getCustomEditor메소드가 정의되었더라도 사용자 편집기를 팝업하지 않을 것이다.

그림 8-24 : 배열을 편집하기 위한 사용자 편집기 다이얼로그

예제 8-11 : InverseEditor.java

```
import java.awt.*;
import java.beans.*;

public class InverseEditor extends PropertyEditorSupport
{  public Component getCustomEditor()
   {   return new InverseEditorPanel(this);
   }

   public boolean supportsCustomEditor()
```

```
    {   return true;
    }

   public boolean isPaintable()
    {   return true;
    }

   public void paintValue(Graphics g, Rectangle box)
    {   boolean isInverse = ((Boolean)getVa lue()).booleanValue();
       String s = isInverse ? "Inverse" : "Normal";
       g.setColor(isInverse ? Color.black : Color.white);
       g.fillRect(box.x, box.y, box.width, box.height);
       g.setColor(isInverse ? Color.white : Color.black);
         FontMetrics fm = g.getFontMetrics();
       int w = fm.stringWidth(s);
       int x = box.x;
      if (w < box.width) x += (box.width - w) / 2;
      int y = box.y + (box.height - fm.getHeight()) / 2
            + fm.getAscent();
       g.drawString(s, x, y);
    }

    public String getAsText()
    {   return null;
    }
}
```

예제 8-12 : InverseEditorPanel.java

```
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.lang.reflect.*;
import java.beans.*;
```

```
import javax.swing.*;

public class InverseEditorPanel extends JPanel
{   public InverseEditorPanel(PropertyEditorSupport ed)
    {   editor = ed;
        ButtonGroup g = new ButtonGroup();
         boolean isInverse
              = ((Boolean)editor.getValue()).booleanValue();
        normal = new JCheckBox("Normal", !isInverse);
        inverse = new JCheckBox("Inverse", isInverse);

         g.add(normal);
         g.add(inverse);
         add(normal);
         add(inverse);

        ActionListener buttonListener =
             new ActionListener()
             {   public void actionPerformed(ActionEvent event)
                 {   editor.setValue(new Boolean(inverse.isSelected()));
                     editor.firePropertyChange();
                 }
              };

         normal.addActionListener(buttonListener);
         inverse.addActionListener(buttonListener);
     }

    private JCheckBox normal;
    private JCheckBox inverse;
     PropertyEditorSupport editor;
}
```

예제 8-13 : DoubleArrayEditor.java

```
import java.awt.*;
```

```
import java.beans.*;

public class DoubleArrayEditor extends PropertyEditorSupport
{   public Component getCustomEditor()
    {   return new DoubleArrayEditorPanel(this);
     }

    public boolean supportsCustomEditor()
    {   return true;
     }

    public boolean isPaintable()
    {   return true;
     }

   public void paintValue(Graphics g, Rectangle box)
   {   double[] values = (double[]) getValue();
       String s = "";
      for (int i = 0; i < 3; i++)
        {   if (values.length > i) s = s + values[i];
          if (values.length > i + 1) s = s + ", ";
         }
      if (values.length > 3) s += "...";

         g.setColor(Color.white);
       g.fillRect(box.x, box.y, box.width, box.height);
         g.setColor(Color.black);
       FontMetrics fm = g.getFontMetrics();
       int w = fm.stringWidth(s);
       int x = box.x;
      if (w < box.width) x += (box.width - w) / 2;
      int y = box.y + (box.height - fm.getHeight()) / 2
            + fm.getAscent();
        g.drawString(s, x, y);
     }
```

```
   public String getAsText()
    {   return null;
    }
}
```

예제 8-14 : DoubleArrayEditorPanel.java

```
import java.awt.*;
import java.awt.event.*;
import java.text.*;
import java.lang.reflect.*;
import java.beans.*;
import javax.swing.*;
import javax.swing.event.*;

public class DoubleArrayEditorPanel extends JPanel
{   public DoubleArrayEditorPanel(PropertyEditorSupport ed)
    {   editor = ed;
        setArray((double[])ed.getValue());

        setLayout(new GridBagLayout());
       GridBagConstraints gbc = new GridBagConstraints();

        gbc.weightx = 100;
        gbc.weighty = 0;
        gbc.fill = GridBagConstraints.HORIZONTAL;

       add(sizeField, gbc, 0, 0, 1, 1);
       add(valueField, gbc, 0, 1, 1, 1);

        gbc.fill = GridBagConstraints.NONE;

       add(sizeButton, gbc, 1, 0, 1, 1);
       add(valueButton, gbc, 1, 1, 1, 1);
```

```
        sizeButton.addActionListener(
            new ActionListener()
            {   public void actionPerformed(ActionEvent event)
                {   changeSize();
                }
            });

        valueButton.addActionListener(
            new ActionListener()
            {   public void actionPerformed(ActionEvent event)
                {   changeValue();
                }
            });

        gbc.weighty = 100;
        gbc.fill = GridBagConstraints.BOTH;

       add(new JScrollPane(elementList), gbc, 0, 2, 2, 1);

        elementLis t.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        elementList.addListSelectionListener(
            new ListSelectionListener()
            {   public void valueChanged(ListSelectionEvent event)
                {   int i = elementList.getSelectedIndex();
                    if (i < 0) return;
                     valueField.setText("" + array[i]);
                }
            });

        elementList.setModel(model);
        elementList.setSelectedIndex(0);
    }

public void add(Component c, GridBagConstraints gbc,
```

```
      int x, int y, int w, int h)
{   gbc.gridx = x;
    gbc.gridy = y;
    gbc.gridwidth = w;
    gbc.gridheight = h;
     add(c, gbc);
}

public void changeSize()
{    fmt.setParseIntegerOnly(true);
    int s = 0;
     try
    {   s = fmt.parse(sizeField.getText()).intValue();
        if (s < 0)
            throw new ParseException("Out of bounds", 0);
     }
    catch(ParseException e)
    {   JOptionPane.showMessageDialog(this, "" + e,
              "Input Error", JOptionPane.WARNING_MESSAGE);
         sizeField.requestFocus();
         return;
     }
   if (s == array.length) return;
    setArray((double[])arrayGrow(array, s));
     editor.setValue(array);
     editor.firePropertyChange();
}

public void changeValue()
{   double v = 0;
     fmt.setParseIntegerOnly(false);
     try
    {   v = fmt.parse(valueField.getText()).doubleValue();
     }
    catch(ParseException e)
    {   JOptionPane.showMessageDialog(this, "" + e,
```

```
            "Input Error", JOptionPane.WARNING_MESSAGE);
         valueField.requestFocus();
         return;
      }
    int currentIndex = elementList.getSelectedIndex();
     setArray(currentIndex, v);
     editor.firePropertyChange();
 }

static Object arrayGrow(Object a, int newLength)
{   Class cl = a.getClass();
    if (!cl.isArray()) return null;
    Class componentType = a.getClass().getComponentType();
    int length = Array.getLength(a);

    Object newArray = Array.newInstance(componentType,
         newLength);
     System.arraycopy(a, 0, newArray, 0,
        Math.min(length, newLength));
     return newArray;
 }

 public double[] getArray()
 {   return (double[])array.clone();
 }

 public void setArray(double[] v)
{   if (v == null) array = new double[0];
    else array = v;
     model.setArray(array);
    sizeField.setText("" + array.length);
    if (array.length > 0)
    {   valueField.setText("" + array[0]);
         elementList.setSelectedIndex(0);
     }
     else
```

```
            valueField.setText("");
    }

    public double getArray(int i)
    {   if (0 <= i && i < array.length) return array[i];
        return 0;
    }

    public void setArray(int i, double value)
    {   if (0 <= i && i < array.length)
        {   model.setValue(i, value);
            elementList.setSelectedIndex(i);
            valueField.setText("" + value);
        }
    }

    private PropertyEditorSupport editor;
    private double[] array;
    private NumberFormat fmt = NumberFormat.getNumberInstance();
    private JTextField sizeField = new JTextField(4);
    private JTextField valueField = new JTextField(12);
    private JButton sizeButton = new JButton("Resize");
    private JButton valueButton = new JButton("Change");
    private JList elementList = new JList();
    private DoubleArrayListModel model = new DoubleArrayListModel();
}

class DoubleArrayListModel extends AbstractListModel
{   public int getSize()
    {   return array.length;
    }

    public Object getElementAt(int i)
    {   return "[" + i + "] " + array[i];
    }
```

```
  public void setArray(double[] a)
  {  int oldLength = array == null ? 0 : array.length;
      array = a;
     int newLength = array == null ? 0 : array.length;
     if (oldLength > 0) fireIntervalRemoved(this, 0, oldLength);
     if (newLength > 0) fireIntervalAdded(this, 0, newLength);
   }

  public void setValue(int i, double value)
   {  array[i] = value;
      fireContentsChanged(this, i, i);
   }

   double[] array;
}
```

요약:

여러분이 작성하는 모든 속성 편집기를 위해, 여러분은 속성 값을 출력하고 편집하기 위한 세가지 방법중의 하나를 선택해야 한다.

- **텍스트 문자열(** getAsText **와** setAsText **정의)을 사용**
- **선택 필드(**getAsText **와** setAsText **그리고** getTags **정의)를 사용**
- **그래픽하게, 그리는 방법(** isPaintable, paintValue, sipportCustomEditor **그리고** getCustomEditor**를 정의**)

여러분은 챠트 빈에서 새가지 경우 모두에 대한 예제를 살펴보았다.

마지막으로, 몇몇 속성 편집기들은 getJavaInitializationString**이라 불리는 메소드를 지원하길 원할 것이다. 이 메소드로, 여러분은 자동 코드 생성을 가능하게 하는 속성을 설정하는 자바 프로그래밍 언어 코드를 빌더 도구에게 제공할 수 있다.**

java.beans.PropertyEditorSupport

- boolean isPaintable()
  만일 클래스가 속성을 출력하기 위해 paintValue **메소드를 사용한다면 이 메소드를** true**를 반환하도록 오버라이드한다.**

- void paintValue(Graphics g, Rectangle box)

속성 시트를 위해 사용하는 컴포넌트상의 특정 위치에 그래픽 컨텍스트에 그려서 값을 표현하기 위해 이 메소드를 오버라이드한다.

전달인자: g 그래픽 객체

box 값을 그리는 속성 시트 컴포넌트상에 표현되는 사각형 객체

- boolean supportsCustomEditor()

  속성 편집기가 사용자 편집기를 가지고 있다면 true를 반환하게 끔 이 메소드를 오버라이드한다.

- component getCustomEditor()

  속성 값을 편집하기 위한 커스터마이징된 GUI를 포함하는 컴포넌트들을 반환하게 끔 이 메소드를 오버라이드한다.

- String getJavaInitializationString()

  속성 값을 초기화하는 코드를 생성하기 위해 사용죄어지는 자바 프로그래밍 언어 코드 문자열을 반환하도록 이 메소드를 오버라이드한다. 가령, “0” , “new Color(64, 64, 64)” .

명명 규칙을 넘어-BeanInfo 클래스 생성하기

여러분은 이미 여러분이 빈의 멤버들을 위한 표준 명명 규칙을 사용한다면 빌더 도구가 여러분의 빈의 속성들과 이벤트들, 메소드들을 결정하기 위해 리플렉션을 사용할 수 있다는 것을 살펴 보았다. 이 과정은 빈 프로그램을 쉽게 시작하게 해주지만 결국에는 다소 제한적이다. 여러분의 빈들이 복잡해 짐에 따라 명명 패턴과 리플렉션 같은 빈의 기능들은 손쉽게 들어나지 않을 것이다. (항상 모든 언어들에서 GUI 빌더들을 위한 토대로서 영어 명명 패턴을 사용하는 것은 다소 다국적 고객들을 지원하기 위한 정신에 반하는 것인 것처럼 보인다.) 더욱이, 우리가 언급했던 것처럼, 빈 속성들을 노출하는 get/set 쌍을 알리길 원치 않을 것이다.

운좋게도, 자바빈 명세는 빌더에서 사용하기 위한 여러분의 빈에 대한 정보를 저장하기 위한 보다 유연하고 강력한 메커니즘을 가능하게 한다. 빈들의 많은 기능들에 의하면 메커니즘은 이론적으로 단순하다. 그러나 실제 수행하는 것은 단조로울 수 있다. (우리가 챠트 빈 클래스를 위한 속성 편집기를 제공할 때 BeanInfo 클래스의 한가지 기능을 사용한다는 것을 상기하라.)

여러분이 여러분의 빈을 설명하는 해당 인터페이스를 구현할 때, 빌더 도구는 빈이 제공하는 속성들과 이벤트들, 메소드들에 대한 정보를 말해주는 BeanInfo 인터페이스의 메소드들을 살펴볼 것이다. BeanInfo는 명명 패턴의 강압으로부터 여러분을 자유롭게 할 것이다. 다소 아

이러니칼 하게도, 자바빈 명세는 빈에 연관된 BeanInfo 객체에 대한 명명 패턴을 사용할 것을 요구한다. 여러분은 BeanInfo를 빈의 이름에 추가함으로서 빈 인포 클래스의 이름을 명시한다. 가령, ChartBean 클래스에 연관되어 있는 빈 인포 클래스는 ChartBeanBeanInfo라고 명명돼야 한다. 빈 인포 클래스는 빈 그 자체로서 같은 패키지의 일부여야 한다.

노트 : 여러분의 빈에 연관되어 있는 빈 인포내에 여러분이 제공하는 모든 설명들은 빌더가 멤버 이름들에 대한 리플렉션에 의해서 얻는 모든 정보를 오버라이드한다. 더욱이, 여러분이 기능 셋에 대한 정보를 제공한다면 (여러분의 빈이 제공하는 속성들처럼), 여러분은 연관된 빈 인포의 모든 속성들에 대한 정보를 제공해야 한다.

여러분이 이미 보았던 것처럼, 일반적으로 여러분은 BeanInfo 인터페이스를 구현하는 클래스를 스크래치부터 작성하지 않는다. 대신에, 여러분은 아마도 BeanInfo 인터페이스의 모든 메소드들에 대해 비어있는 구현(null을 반환)을 갖는 SimpleBeanInfo라는 편리한 클래스를 다시 찾을 것이다. 이것은 분명 편리하다 – 단지 여러분이 실제 변경하길 원하는 메소드들을 오버라이드한다. 더욱이 이 편리한 클래스는 여러분의 빈을 위해 이미지를 로드하기 위해(아이콘처럼 – 아래 참조) 사용하는 사용할 수 있는 loadImage라 불리는 유용한 메소드를 포함하고 있다. 우리는 우리의 모든 BeanInfo 클래스들의 예제를 위해 SimpleBeanInfo 클래스를 사용한다. 가령, 우리의 ChartBeanBeanInfo 클래스는 다음처럼 시작한다:

```
public class ChartBeanBeanInfo extends SimpleBeanInfo
```

노트 : null을 반환하는 SimpleBeanInfo 클래스의 해당 메소드들은 실제 아주 중요하다. 이것은 해당 기능 셋의 멤버들을 찾기 위한 명명 패턴을 사용하는 방법을 빌더 도구가 아는 실제 방법이다. null 이 아닌 반환 값은 리플렉티브 탐색을 해제한다.

빈 인포 메커니즘으로 여러분이 할 수 있는 일을 경험하기 위해, 사용하기 쉽지만 가장 유용한 BeanInfo 인터페이스내의 메소드로 시작해 보자 : getIcon 메소드는 여러분의 빈에 사용자 아이콘을 제공해준다. 이것은 빌더 도구가 일반적으로 빈을 위한 아이콘을 갖길 원하기 때문에 유용하다. 빈박스에서, 아이콘은 툴박스내에 빈 이름의 왼쪽에 나타난다 – 그림8-3참조. 실제, 여러분은 별도의 아이콘 비트맵들을 명시할 수 있다. BeanInfo 인터페이스는 표준 크기를 위해 4개의 상수를 가지고 있다.

```
ICON_COLOR_16x16
```

ICON_COLOR_32x32

ICON_MONO_16x16

ICON_MONO_32x32

다음은 클래스에 아이콘을 추가하는 SimpleBeanInfo 클래스의 loadImage 메소드를 사용하는 방법에 대한 예제이다.

```
public Image getIcon(int iconType)
  {  String name = "";
     if (iconType == BeanInfo.ICON_COLOR_16x16)
         name = "COLOR_16x16";
      else if (iconType == BeanInfo.ICON_COLOR_32x32)
         name = "COLOR_32x32";
     else if (iconType == BeanInfo.ICON_MONO_16x16)
         name = "MONO_16x16";
     else if (iconType == BeanInfo.ICON_MONO_32x32)
         name = "MONO_32x32";
      else return null;
     return loadImage("ChartBean_" + name + ".gif");
  }
```

우리는 다음처럼 분명하게 명명된 이미지 파일들을 갖어야 한다.

ChartBean_COLOR_16x16.gif

ChartBean_COLOR_32x32.gif

FetureDescriptor 객체들

BeanInfo 클래스의 보다 향상된 기능들을 사용하는 열쇠는 FeatureDescriptor 클래스와 이 클래스의 다양한 하위 클래스이다. 이 클래스의 이름이 암시하듯이, FeatureDescriptor 객체는 기능에 대한 정보를 제공한다. 기능들의 예제는 속성들, 이벤트들, 메소드들 기타 등등 이다. 보다 자세히 살펴보면, FeatureDescriptor 클래스는 모든 설명자(descriptor)들의 상위 클래스이다. 그리고 모든 기능들의 설명을 시도할 때 처리해야 할 필요가 있는 공통의 기능들을 모아 놓았다. 가령, 기능의 이름은 getName 메소드를 통해 얻는다. 이 메소드는 상위 클래스에 있기 때문에 설명이 무엇인지 상관없이 모든 기능 설명자들에 대해 동작한다. 다음은 FeatureDescriptor 클래스의 하위 클래스들이다.

- BeanDescriptor
- EventSetDescriptor
- MethodDescriptor
- ParameterDescriptor
- PropertyDescriptor (하위에 IndexedPropertyDescriptor 클래스를 갖는다.)

이들 클래스들은 기본적으로 모두 똑같이 동작한다. 여러분은 설명하려고 시도하는 각 멤버들을 위한 설명자 객체를 생성한다. 그리고 여러분은 배열에 기능 셋의 모든 설명자들을 수집한다. 그리고 BeanInfo 메소드들중의 하나의 반환값으로 이를 반환한다.

예를 들면, 이벤트 셋들을 위한 리플렉션을 해제하기 위해, 여러분은 여러분의 빈 인포 클래스에 EventSetDescriptor 객체들의 배열을 반환할 것이다.

```
class MyBeanBeanInfo extends SimpleBeanInfo
{       public EventSetDescriptor[] getEventSetDescriptor()
        {       …
        }
        …
}
```

다음으로, 여러분은 해당 배열내에 계속해서 다양한 EventSetDescriptor 객체들을 모두 구성할 것이다. 일반적으로, 다양한 종류의 EventSetDescriptor 객체들에 대한 모든 생성자들은 ㄱ샅은 방법으로 작동한다. 특히, 이벤트들에 대해 가장 일반적인 생성자는 다음을 취한다:

- 해당 이벤트를 갖고 있는 빈 클래스
- 이벤트의 베이스 이름
- 이벤트에 해당하는 EventListener 인터페이스의 클래스
- 이벤트에 의해서 트리거되는 EventListener 인터페이스에 명시된 메소드들

다른 생성자들은 EventListener객체들을 추가하고 제거하기 위해 사용되어지는 빈의 메소드들을 명시하게 해준다.

이 모든 것의 좋은 예제는 BDK에서 제공하는 ExplicitButtonBean과 연관되어 있는 BeanInfo 클래스내에서 발견되어 질 수 있다. 해당 빈을 위해 필요한 이벤트 설명자들을 생성하기 위한 코드를 분석해 보자. ExplicitButtonBean은 두개의 이벤트를 발생한다: 버튼이 눌러졌을 때와 버튼의 상태가 변경됐을 때. 다음은 이러한 두개의 이벤트들에 연관되어 있는

EventSetDescriptor 객체들을 구성하는 방법이다:

```
EventSetDescriptor push = new EventSetDescriptor(beanClass,
        “actionPerformed”,
        java.awt.event.ActionListener.class,
        “actionPerformed” );

EventSetDescriptor changed = new EventSetDescriptor(beanClass,
        “propertyChange”,
        java.beans.PropertyChangeListener.class,
        “propertyChange” );
```

다음 단계는 EventSetDescriptor 를 위한 이벤트들의 다양한 디스플레이 이름들을 설정하는 것이다. BeanInfo 클래스의 ExplicitButton을 위한 코드에서, 다음에 의해서 수행된다.

```
push.setDisplayName(“button push”);
changed.setDisplayName(“bond property change”);
```

실제로, 기능 설명자 객체들을 위한 모든 생성자들이 IntroSpectionException을 투구할 수 있기 때문에 위의 코드 조각들보다 필요한 EventSetDescriptor 객체들을 생성하기 위한 코드가 추가적으로 필요하다. 그래서 여러분은 다음의 코드처럼 실제 try/catch 블록내에 설명자들의 배열을 구성해야 한다.

```
public EventSetDescriptor[] getEventSetDescriptors()
{       try
        {       EventSetDescriptor push = new EventSetDescriptor(beanClass,
                “actionPerformed”,
                java.awt.event.ActionListener.class,
                “actionPerformed” );

                EventSetDescriptor changed = new EventSetDescriptor(beanClass,
                “propertyChange”,
                java.beans.PropertyChangeListener.class,
                “propertyChange” );
```

```
                push.setDisplayName("button push");
                changed.setDisplayName("bond property change");
                return new EventSetDescriptor[] { push, changed };
        }catch( IntrospectionException e)
        {       throw new Error(e.toString() );
        }
}
```

이 특정한 이벤트 셋 설명자들은 이벤트 설명자들이 표준 명명 패턴과 다르기 때문에 필요하다. 특히, 다음 두개의 차이점이 존재한다:

- **리스너 클래스들은 빈** + Listener **의 이름이 아니다.**
- **디스플레이 이름들은 이벤트 이름들과 같지 않다.**

요약 : 여러분의 빈의 모든 기능이 표준 명명 패턴과 다를 때, 여러분은 다음을 수행해야 한다:

- **해당 셋(** 이벤트들, 속성들, 메소드들**)내에 모든 기능들을 위한 기능 설명자를 생성한다.**
- **적합한** BeanInfo **메소드내의 설명자의 배열을 반환한다.**

java.beans.BeanInfo

- EventSetDescriptor[] getEventSetDescriptors()
- MethodDescriptor[] getMethodDescriptors()
- PropertyDescriptor[] getPropertyDescriptors()
  명시된 설명자 객체들의 배열을 반환한다. null **반환은 빌더에게 명명 규칙을 사용하고 멤버를 찾기위해 리플렉션을 사용하는 신호이다.** GetPropertyDescriptors **메소드는 일반과 색인된 속성 설명자들의 혼합을 반환한다. 만일 특정** PropertyDescriptor**가** IndexedPropertyDescriptor**인지 검사하려면** instanceof**를 사용한다.**

- Image getIcon(int iconType)
  툴박스, 툴 바 같은 곳에서 빈을 표현하기 위해 사용될 수 있는 이미지 객체를 반환한다. 여기에는 표준 아이콘 타입을 위한 4**개의 상수가 있다.**
- int getDefaultEventIndex()
- int getDefaultPropertyIndex()
  빈은 기본 이벤트나 속성을 갖을 수 있다. 이들 메소드들은 모두 기본 멤버로서 사용하는 설명자 배열의 항목을 명시하는 배열 색인을 반환하거나 기본이 존재하지

않으면 –1을 반환한다. 빈 빌더 환경은 비쥬얼하게 기본 기능을 강화할 수 있다. 예를 들면, 기능들의 목록의 첫번째 위치시키거나 볼드체로 이름을 출력하는 것에 의해서 기본 기능을 강화할 수 있다.

- BeanInfo[] getAdditionalBeanInfo()
  BeaniInfo 객체의 배열이나 null을 반환한다. 여러분이 다른 빈들을 위해 BeanInfo 클래스들로부터 나오는 여러분의 빈에 대한 몇몇 정보를 원할 때 이 메소드를 사용한다. 가령, 여러분의 빈이 다른 빈들의 합계를 수집한다면 이 메소드를 사용한다.

java.beans.SimpleBeanInfo

- Image loadImage(String resourceName)
  자원과 연관된 이미지 객체를 반환한다. 현재 오직 GIF 파일만 지원한다.
  전달인자: resourceName 경로 이름(현재 클래스를 포함하고 있는 상대 디렉토리)

java.beans.FeatureDescriptor

- String getName()
  멤버에 대해 빈의 코드내에서 사용되는 이름을 반환한다.
- void setName(String name)
  해당 기능(feature)의 이름을 설정한다.
  전달인자: name 기능의 이름

- String getDisplayName()
  해당 기능의 로컬화된 디스플레이 이름을 반환한다. 기본 값은 getName에 의해서 반환되는 값이다. 그러나, 현재 다중 로케일로 기능 이름들을 제공하기 위한 지원은 존재하지 않는다.

- void setDisplayName(String displayName)
  기능의 로컬화된 디스플레이 이름을 설정한다.
  전달인자: displayName 사용할 이름

- String getShortDescription()
  빌더 도구에서 해당 기능에 대한 짧은 설명을 제공하기 위해 사용할 수 있는 로컬화된 문자열을 반환한다. 기본값은 getDisplayName의 반환 값이다.

- void setShortDescription(String text)

기능을 설명하는 설명 문자열(일반적으로 40자 이내)을 설정한다.
전달인자: text 해당 기능과 관련이 있는 로컬화된 짧은 설명

- void setValue(String attributeName, Object value)
  명명된 속성을 개항 기능에 연관시킨다.
  전달인자: attributeName 여러분이 설정한 값을 갖는 속성의 이름

- Object getValue(String attributeName)
  주어진 이름의 속성 값을 얻는다.
  전달인자: attributeName 추출할 속성의 이름

- Enumeration attributeNames()
  setValue로 등록된 모든 속성들의 이름들을 포함하고 있는 열거 객체를 반환한다.

- void setExpert(boolean b)
  빌더가 사용자로부터 기능을 숨길것인지 결정하기 위해 사용할 수 있는 전문가 플레그를 제공하게 해준다.(모든 빌더가 이 기능을 제공하지는 않는다.)
  전달인자: b 만일 이 기능이 오직 전문가들에 의해서만 사용되고자 한다면 true를 설정한다.

- boolean isExpert()
  만일 이 기능이 전문가들에 의해서 사용되기 위해 마크되어 있다면 true를 반환한다.

- void setHidden(boolean b)
  오직 빌더 도구에 의해서 사용되기 위한 기능을 표시한다.
  전달인자: b 만일 여러분이 이 기능을 숨기길 원한다면 true로 설정

- boolean isHidden()
  만일 빌더의 사용자가 이 기능을 볼 수 없지만 빌더 도구가 이 기능을 인식할 필요가 있을 때 true를 반환환다.

java.beans.EventSetDescriptor

- EventSetDescriptor(Class sourceClass, String eventSetName, Class listener, String listenerMethod)
  EventSetDescriptor를 구성한다. 이 생성자는 이벤트 클래스의 이름들과 이벤트 리스

너를 추가하고 제거하는 메소드의 이름들이 표준 패턴을 따른다고 가정한다. 만일 내성(인트로스펙션, introspection)동안 에러가 발생하면 IntrospectionException을 발생한다.

전달인자:	sourceClass	이벤트를 발생하는 클래스
	eventSetName	이벤트의 이름
	listener	전달될 이벤트들에 대한 리스너 인터페이스
	listenerMethod	이벤트가 리스너에게 전달됐을 때 발생하는 메소드

- EventSetDescriptor(Class sourceClass, String eventSetName, Class listener, String[] listenerMethods, String addListenerMethod, String removeListenerMethod)
  EventSetDescriptor의 가장 일반적인 생성자이다. 만일 내성(인트로스펙션, introspection)동안 에러가 발생하면 IntrospectionException을 발생한다.

전달인자:	sourceClass	이벤트를 발생하는 클래스
	eventSetName	이벤트의 이름
	listener	전달될 이벤트들에 대한 리스너 인터페이스
	listenerMethod	이벤트가 리스너에게 전달됐을 때 발생하는 메소드

- Method getAddListenerMethod()
  리스너를 등록하기 위해 사용되는 메소드를 반환한다.
- Method getRemoveListenerMethod()
  해당 이벤트에 대한 등록된 리스너를 제거하기 위해 사용되는 메소드를 반환한다.
- Method[] getListenerMethods()
- MethodDescriptor[] getListenerMethodDescriptors()
  리스너 인터페이스내에 트리거된 메소드들에 대한 Method나 MethodDescriptor 객체들의 배열을 반환한다.
- Class getListenerType()
  해당 이벤트와 연관되어 있는 목표 리스너 인터페이스를 위한 Class 객체를 반환한다.
- void setUnicast(boolean b)
  만일 해당 이벤트가 오직 하나의 리스너에게 만 전파될 수 있다면 true로 설정한다.
- boolean isUnicast()
  해당 이벤트 셋이 유니캐스트이면 true로 설정한다.(기본은 false이다.)

java.beans.PropertyDescriptor

- PropertyDescriptor(String PropertyName, Class beanClass)
- PropertyDescriptor(String PropertyName, Class beanClass , String getMethod, String setMethod)

  PropertyDescriptor객체를 생성한다. 만일 내성(인트로스펙션, introspection)동안 에러가 발생하면 IntrospectionException을 발생한다. 첫번째 생성자는 get과 set 메소드들의 이름에 대한 표준 명명 규칙을 따른다고 가정한다.

  전달인자:

PropertyName	속성의 이름
beanClass	설명된 빈을 위한 Class 객체
getMethod	get 메소드의 이름
setMethod	set 메소드의 이름

- Class getPropertyType()

  속성 타입에 대한 Class 객체를 반환한다.
- Method getReadMethod()

  get 메소드를 반환한다.
- Method getWriteMethod()

  set 메소드를 반환한다.
- void setBound(Boolean b)

  만일 해당 속성의 값이 변경됐을 때 PropertyChangeEvent를 발생시키려면 true를 설정한다.
- boolean isBound()

  해당 속성이 바운드 속성이면 true를 반환한다.
- void setConstrained(boolean b)

  만일 해당 속성의 값이 변경됐을 때 VetoableChangeEvent를 발생시키려면 true를 설정한다.
- boolean isConstrained()

  해당 속성이 제약 속성이면 true를 반환한다.

java.beans.IndexedPropertyDescriptor

- IndexedPropertyDescriptor(String PropertyName, Class beanClass)
- IndexedPropertyDescriptor(String PropertyName, Class beanClass , String getMethod, String setMethod , String IndexedGetMethod , String IndexedSetMethod)

  색인 속성을 위한 IndexedPropertyDescriptor를 생성한다. 만일 내성(인트로스펙션,

introspection)동안 에러가 발생하면 IntrospectionException을 발생한다. 첫번째 생성자는 get과 set 메소드들의 이름에 대한 표준 명명 규칙을 따른다고 가정한다.

전달인자:	PropertyName	속성의 이름
	beanClass	설명된 빈을 위한 Class 객체
	getMethod	get 메소드의 이름
	setMethod	set 메소드의 이름

- Class getIndexedPropertyType()
  속성의 색인된 값들의 타입을 설명하는 자바 플랫폼 클래스를 반환한다. 즉, 색인된 get 메소드의 반환 타입이다.
- Method getIndexedReadMethod()
  색인된 get 메소드를 반환한다.
- Method getIndexedWriteMethod()
  색인된 set 메소드를 반환한다.

java.beans.MethodDescriptor

- MethodDescriptor(Method method)
- MethodDescriptor(Method method , ParameterDescriptor[]ParameterDescriptors)
  관련된 전달인자를 갖는 주어진 메소드에 대한 메소드 설명자를 구성한다. 만일 내성(인트로스펙션, introspection)동안 에러가 발생하면 IntrospectionException을 발생한다.

전달인자:	method	메소드 객체
	ParameterDescriptors	메소드에 대한 전달인자들을 설명하는 전달인자 설명자의 배열

- Method getMethod()
  해당 메소드에 대한 메소드 객체를 반환한다.
- ParameterDescriptor[] getParameterDescriptors()
  메소드들의 전달인자들에 대한 전달인자 설명자 객체의 배열을 반환한다.

java.beans.ParameterDescriptor

- ParameterDescriptor()
  새로운 전달인자 설명자 객체를 생성한다. 전달인자 설명자들은 FeatureDescriptor 상위 클래스내에 저장되어 있는 정보를 제공한다.

커스터마이져들

아무리 복잡하더라도, 속성 편집기는 사용자에게 한번에 하나의 속성을 설정하게 해주는 책임이 있다. 특히, 어떤 빈의 속성들이 서로 연관되어 있다면 동시에 다중 속성들을 편집하는 방법을 보다 편하게 사용자에게 제공할 것이다. 이 기능을 가능하게 하기 위해, 여러분은 다중 속성 편집기들 대신에 *커스터마이저*를 제공한다.

이 절을 위한 예제 프로그램에서, 우리는 챠트 빈을 위한 커스터마이저를 개발했다. 커스터마이져는 여러분에게 한번에 챠트 빈의 몇몇 속성들을 설정해준다. 그리고 챠트를 위한 데이터를 읽기 위한 파일을 지정하게 해준다.

그림 8-25 : 챠트빈을 위한 커스터마이져

여러분의 빈에 커스터마이져를 추가하기 위해, 여러분은 다음의 예제에서 처럼 BeanInfo 클래스를 제공해야 하고 getBeanDescriptor 메소드를 오버라이드해야 한다.

```
public BeanDescriptor getBeanDescriptor()
{       return new BeanDescriptor(ChartBean.class , ChartBeanCustomizer.class);
}
```

같은 모델로 여러분의 커스터마이져들을 위한 일반적인 절차는 다음과 같다.

1. 여러분의 빈을 위한 새로운 BeanDescriptor 객체를 반환하도록 getBeanDescriptor 메소드를 오버라이드한다.
2. BeanDescriptor 객체를 위한 생성자의 두번째 전달인자로서 커스터마이져 클래스를 명시한다.

여러분은 커스터마이져 클래스에 대해 어떠한 명명 패턴을 준수할 필요가 없다는 것을 기억하라. 빌더는 다음에 의해서 이를 위치시킨다.

1. 관련된 BeanInfo 클래스를 찾는다.
2. getBeanDescriptor 메소드를 호출한다.
3. getCustomizerClass 메소드를 호출한다.

(그럼에도 불구하고, BeanNameCustomizer 처럼 커스터마이져에 대한 이름을 작명하는 것이 일반적인 방법이다.)

예제 8-15는 ChartBeanCustomizer를 참조하는 ChartBeanBeanInfo 클래스에 대한 코드이다. 여러분은 다음 절에서 커스터마이져를 구현하는 방법을 보게 될 것이다.

예제 8-15 : ChartBeanBeanInfo.java

```
import java.awt.*;
import java.beans.*;

public class ChartBeanBeanInfo extends SimpleBeanInfo
{  public BeanDescriptor getBeanDescriptor()
   {  return new BeanDescriptor(ChartBean.class,
            ChartBeanCustomizer.class);
    }

   public Image getIcon(int iconType)
   {  String name = "";
      if (iconType == BeanInfo.ICON_COLOR_16x16)
          name = "COLOR_16x16";
       else if (iconType == BeanInfo.ICON_COLOR_32x32)
          name = "COLOR_32x32";
      else if (iconType == BeanInfo.ICON_MONO_16x16)
          name = "MONO_16x16";
      else if (iconType == BeanInfo.ICON_MONO_32x32)
          name = "MONO_32x32";
       else return null;
      return loadImage("ChartBean_" + name + ".gif");
    }
}
```

java.beans.BeanInfo

- BeanDescriptor getBeanDescriptor()
  빈의 기능들을 설명하는 BeanDescriptor**를 반환한다**.

java.beans.BeanInfo

- BeanDescriptor(Class beanClass, Class customizerClass)
  커스터마이저를 갖고 있는 빈에 대한 BeanDescriptor **객체를 구성한다**.
  전달인자: beanClass **빈을 위한** Class **객체**
  customizerClass **빈의 커스터마이저를 위한** Class **객체**
- Class getBeanClass()
  빈을 정의하는 Class **객체를 반환한다**.
- Class getCustomizerClass()

빈의 커스터마이져를 정의하는 Class 객체를 반환한다.

커스터마이져 클래스 작성하기

여러분이 작성하는 모든 커스터마이져 클래스는 Customizer 인터페이스를 구현해야 한다. 이 인터페이스에는 오직 3개의 메소드만이 존재한다:

- 커스터마이징된 빈을 명시하는 전달인자를 취하는 setObject 메소드
- 커스터마이져내에서 속성이 변경했을 때를 알려주는 리스너들의 컬렉션을 관리하는 addPropertyChangeListener와 removePropertyChangeListener 메소드

단지 사용자가 커스터마이징 과정의 끝에 있을 때가 아니라, 사용자가 속성 값들을 변경할 때마다 PropertyChangeEvent를 브로드캐스팅하여 목표 빈의 외관을 갱신하는 것은 좋은 생각이다.

속성 편집기와 달리, 커스터마이져들은 자동적으로 출력하지 못한다. 빈박스에서, 여러분은 빈의 커스터마이져를 팝업하기 위해 Edit->Customizer를 선택해야 한다. 이 시점에서, 빈박스는 전달인자로서 커스터마이징된 빈을 취하는 커스터마이져의 setObject 메소드를 호출할 것이다. 여러분의 커스터마이져는 빈의 인스턴스에 실제 연결되기 전에 생성돼야 한다는 것에 주목하라. 그러므로, 여러분은 커스터마이져내에서 빈의 상태에 대한 어떠한 정보도 가정할 수 없다. 그리고 여러분은 전달인자를 갖지않는 기본 생성자를 제공해야 한다.

커스터마이져 클래스를 작성하는 데에는 3개의 부분이 존재한다.

1. 비쥬얼 인터페이스 작성
2. setObject 메소드내에서 커스터마이져를 초기화
3. 사용자가 인터페이스내에서 속성들을 변경할 때 속성 변경 이벤트들을 발생시켜 빈을 갱신한다.

정의에 따르면, 커스터마이져 클래스는 비쥬얼하다. 그러므로 Component나 JPanel처럼 Component의 하위클래스 확장해야 한다. 커스터마이져는 기본적으로 사용자에게 많은 옵션들을 제공하기 때문에 종종 탭이 가능한 패인 인터페이스를 사용하는 것이 편리하다. 우리는 이 접근 방법을 사용하고 JtabbedPane 인터페이스를 확장한 커스터마이져를 갖는다.

주의: 우리는 빈박스의 커스터마이져 다이얼로그내에 JtabbedPane을 위치시키는 것에 감사한다. 빈박스는 이전 AWT 컴포넌트들을 사용한다. 이벤트들은 AWT와 스윙 컴포넌트들 사이를 복잡하게 한다. 그리고 탭을 클릭해도 패인들이 펼쳐지지 않는다. 많은 테스트틀 통해, 우리는 탭 클릭을 트랩라고 validate를 호출하는 것이 이 문제의 해결책인 것을 발견하였다.

시간이 지난 다음, 빈 환경이 스윙을 사용할 때 이 문제는 사라질 것이다.

커스터마이져는 3개의 패인들에 정보를 수집한다:

- 그래프 색상과 전환 모드
- 제목과 제목 위치
- 데이터 위치

물론, 이러한 종류의 사용자 인터페이스를 개발하는 것은 코딩하기 아주 지겹다 – 우리의 예제는 생성자에 이를 설정하기 위해 100줄이 넘는 코딩을 할 애 했다. 그러나 이 작업은 오직 일반적인 스윙 프로그래밍 기술을 요구한다. 그리고 우리는 여기서 자세한 것에 대해서는 고민하지 않는다.

명심해야 할 한가지 방법이 있다. 여러분은 종종 커스터마이져내에서 속성 값들을 편집할 필요가 있다. 특정 클래스의 속성 값을 설정하기 위한 새로운 인터페이스를 구현하는 것보다, 기존의 속성 편집기를 위치시키거나 여러분의 사용자 인터페이스내에 이를 추가할 수 있다! 예를 들면, 우리의 ChartBean 커스터마이져에서 우리는 그래프 색상을 설정할 필요가 있다. 빈박스가 완벽하게 색상들을 위한 좋은 속성 편집기를 갖고 있다는 사실을 알고 있기 때문에, 우리는 다음처럼 이를 위치시킨다:

```
PropertyEditor colorEditor = PropertyEditorManager.findEditor(Color.Class);
```

그럼 다음, 우리는 색상을 설정하기 위한 사용자 인터페이스를 포함하고 있는 컴포넌트를 얻기위해 getCustomEditor를 호출한다.

```
Component colorEditorComponent = colorEditor.getCustomerEditor();
// now adfd thus component to the UI
```

일단 모든 컴포넌트들의 레이아웃을 마치면, setObject 메소드로 그들의 값들을 초기화한다. setObject 메소드는 커스터마이져가 출력될 때 호출된다. 이 메소드의 전달인자는 커스터마이징된 빈이다. 처리를 위해, 우리는 빈 참조자를 저장한다- 우리는 나중에 빈이 속성 변화를 알리기 위해 이를 필요로 한다. 그런 다음, 우리는 각 사용자 인터페이스 컴포넌트를 초기화한다. 다음은 이러한 초기화를 수행하는 챠트 빈 커스터마이져의 setObject 메소드의 일부이다.

```
public void setObject(Object obj)
{       bean = (ChartBean)obj;
        titleField.setText(bean.getTitle());
        colorEditor.setValue(bean.getGraphColor());
```

```
        …
    }
```

마지막으로, 우리는 사용자의 활동을 추적하기 위해 이벤트 핸들러들을 훅업한다. 사용자가 컴포넌트의 값을 변경할 때마다, 컴포넌트는 우리의 커스터마이저가 다뤄야 하는 이벤트를 발생시킨다. 이벤트 핸들러는 빈에 있는 속성의 값을 갱신해야 한다. 그리고 또한 (속성 시트 같은) 다른 리스너들이 갱신할 수 있도록 하기위해 PropertyChangeEvent를 발생해야 한다. 이제부터 챠트 빈 커스터마이져에서 두개의 사용자 인터페이스 항목들을 처리하기 위해 이 과정을 따라보자.

사용자가 새로운 제목을 입력했을 때, 우리는 제목 속성을 변경하길 원한다. 우리는 사용자가 제목을 입력하는 텍스트 필드에 DocumentListener를 부착한다.

```
titleField.getDocument().addDocumentListener(
        new DocumentListener()
        {   public void changedUpdate(DocumentEvent evt)
            {   setTitle(titleField.getText());
            }
            public void insertUpdate(DocumentEvent evt)
            {   setTitle(titleField.getText());
            }
            public void removeUpdate(DocumentEvent evt)
            {   setTitle(titleField.getText());
            }
        });
```

세개의 리스너 메소드들은 커스터마이저의 setTitle 메소드를 호출한다. 해당 메소드는 속성 값을 갱신하기 위해 빈을 초루한다. 그리고 속성 변화 이벤트를 발생시킨다. (이 갱신은 오직 바운드되는 속성들을 위해 필요하다.) 다음은 setTitle 메소드를 위한 코드이다.

```
public void setTitle(String newValue)
  {  if (bean == null) return;
     String oldValue = bean.getTitle();
      bean.setTitle(newValue);
     firePropertyChange("title", oldValue, newValue);
   }
```

색상 속성 편집기에서 색상 값이 변했을 때, 우리는 빈의 그래프 색을 갱신하길 원한다. 우리는 속성 편집기에 리스너를 부착하여 생상 변화를 추적한다. 아마 혼란스럽게도, 해당 편집기는 속성 변경 이벤트 또한 보낸다.

```
colorEditor.addPropertyChangeListener(
        new PropertyChangeListener()
        {  public void propertyChange(PropertyChangeEvent
              evt)
           {  setGraphColor((Color)colorEditor.getValue());
           }
         });
```

색상 속성 편집기의 색상 값이 변할 때마다, 우리는 커스터마이저의 GraphColor 메소드를 호출한다. 해당 메소드는 빈의 graphColor 속성을 갱신하고 graphColor 속성과 연관되어 있는 다른 속성 변경 이벤트를 발생시킨다.

```
public void setGraphColor(Color newValue)
  {  if (bean == null) return;
     Color oldValue = bean.getGraphColor();
      bean.setGraphColor(newValue);
     firePropertyChange("graphColor", oldValue, newValue);
   }
```

예제 8-16은 차트 빈 커스터마이저의 완전한 코드를 제공한다.

이 특별한 커스터마이저는 단지 빈의 속성들을 설정한다. 일반적으로 커스터마이져는 속성 설정되었든 아니든 상관없이 빈의 모든 메소드들을 호출할 수 있다. 즉, 커스터마이져는 속성 편집기보다 일반적이다. (몇몇 빈은 속성들로 노출되지 않는 기능들을 가지고 있다. 그리고 오직 커스터마이져를 통해 편집될 수 있다.)

예제 8-16 : ChartBeanCustomizer.java

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.io.*;
```

```
import java.text.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

public class ChartBeanCustomizer extends JTabbedPane
    implements Customizer
{   public ChartBeanCustomizer()
   {   data = new JTextArea();
       JPanel dataPane = new JPanel();
        dataPane.setLayout(new BorderLayout());
        dataPane.add(new JScrollPane(data), "Center");
       JButton dataButton = new JButton("Set data");
        dataButton.addActionListener(
            new ActionListener()
           {   public void actionPerformed(ActionEvent event)
                {   setData(data.getText());
                }
            });
       JPanel p = new JPanel();
        p.add(dataButton);
        dataPane.add(p, "South");

       JPanel colorPane = new JPanel();
        colorPane.setLayout(new BorderLayout());

       normal = new JCheckBox("Normal", true);
       inverse = new JCheckBox("Inverse", false);
       p = new JPanel();
        p.add(normal);
        p.add(inverse);
       ButtonGroup g = new ButtonGroup();
        g.add(normal);
        g.add(inverse);
        normal.addActionListener(
            new ActionListener()
```

```
    {   public void actionPerformed(ActionEvent event)
        {   setInverse(false);
        }
     });

  inverse.addActionListener(
     new ActionListener()
    {   public void actionPerformed(ActionEvent event)
        {   setInverse(true);
        }
     });

  colorEditor
     = PropertyEditorManager.findEditor(Color.class);
  colorEditor.addPropertyChangeListener(
     new PropertyChangeListener()
    {   public void propertyChange(PropertyChangeEvent
           evt)
        {   setGraphColor((Color)colorEditor.getValue());
        }
     });

  colorPane.add(colorEditor.getCustomEditor(), "North");
  colorPane.add(p, "South");

 JPanel titlePane = new JPanel();
  titlePane.setLayout(new BorderLayout());

 g = new ButtonGroup();
  position = new JCheckBox[3];
 position[0] = new JCheckBox("Left", false);
 position[1] = new JCheckBox("Center", true);
 position[2] = new JCheckBox("Right", false);

 p = new JPanel();
for (int i = 0; i < position.length; i++)
```

```
{   final int value = i;
      p.add(position[i]);
      g.add(position[i]);
      position[i].addActionListener(
         new ActionListener()
        {   public void actionPerformed(ActionEvent event)
            {   setTitlePosition(value);
            }
         });
 }

titleField = new JTextField();
 titleField.getDocument().addDocumentListener(
     new DocumentListener()
    {   public void changedUpdate(DocumentEvent evt)
          {   setTitle(titleField.getText());
          }
        public void insertUpdate(DocumentEvent evt)
          {   setTitle(titleField.getText());
          }
        public void removeUpdate(DocumentEvent evt)
          {   setTitle(titleField.getText());
          }
     });

 titlePane.add(titleField, "North");
 titlePane.add(p, "South");
 addTab("Color", colorPane);
 addTab("Title", titlePane);
 addTab("Data", dataPane);

addChangeListener( // workaround for a JTabbedPane bug in JDK 1.2
     new ChangeListener()
    {   public void stateChanged(ChangeEvent event)
          {   validate();
          }
```

```
        });
 }

public void setData(String s)
{   StringTokenizer tokenizer = new StringTokenizer(s);

    int i = 0;
    double[] values = new double[tokenizer.countTokens()];
     while (tokenizer.hasMoreTokens())
    {   String token = tokenizer.nextToken();
         try
        {   values[i] = Double.parseDouble(token);
             i++;
         }
        catch (NumberFormatException exception)
         {
         }
     }
     setValues(values);
 }

public void setTitle(String newValue)
{   if (bean == null) return;
    String oldValue = bean.getTitle();
     bean.setTitle(newValue);
    firePropertyChange("title", oldValue, newValue);
 }

public void setTitlePosition(int i)
{   if (bean == null) return;
    Integer oldValue = new Integer(bean.getTitlePosition());
    Integer newValue = new Integer(i);
     bean.setTitlePosition(i);
    firePropertyChange("titlePosition", oldValue, newValue);
 }
```

```
public void setInverse(boolean b)
{   if (bean == null) return;
    Boolean oldValue = new Boolean(bean.isInverse());
    Boolean newValue = new Boolean(b);
      bean.setInverse(b);
     firePropertyChange("inverse", oldValue, newValue);
 }

public void setValues(double[] newValue)
{   if (bean == null) return;
     double[] oldValue = bean.getValues();
      bean.setValues(newValue);
      firePropertyChange("values", oldValue, newValue);
 }

public void setGraphColor(Color newValue)
{   if (bean == null) return;
     Color oldValue = bean.getGraphColor();
      bean.setGraphColor(newValue);
     firePropertyChange("graphColor", oldValue, newValue);
 }

public void setObject(Object obj)
{   bean = (ChartBean)obj;

      data.setText("");
     double[] values = bean.getValues();
   for (int i = 0; i < values.length; i++)
         data.append(values[i] + "\n");

      normal.setSelected(!bean.isInverse());
      inverse.setSelected(bean.isInverse());

      titleField.setText(bean.getTitle());

   for (int i = 0; i < position.length; i++)
```

```
            position[i].setSelected(i == bean.getTitlePosition());

        colorEditor.setValue(bean.getGraphColor());
    }

    public Dimension getPreferredSize()
    {   return new Dimension(200, 120);
    }

    private ChartBean bean;
    private PropertyEditor colorEditor;

    private JTextArea data;
    private JCheckBox normal;
    private JCheckBox inverse;
    private JCheckBox[] position;
    private JTextField titleField;
}
```

java.beans.Customizer

- void setObject(Object bean)
  커스터마이징할 빈을 명시한다.

빈 컨텍스트

이 절에서, 우리는 빈의 환경을 이용하는 빈들을 작성하는 방법을 보여준다. 이것은 다른 빈이나 빈박스에서 제공하는 서비스들과 상호작용하는 빈을 구현하는 데 유용한다. 특히, 여러분이 다른 빈의 임의의 정수 속성 값을 변경할 수 있는 빈을 구현하는 방법과 빈박스에서 제공하는 *메시지 추적 서비스*를 빈에서 사용하는 방법을 보게 될 것이다.

내성(Introspection)의 고급 사용

자바 빈 명세에서, 내성은 어느 빌더 도구가 빈이 어떤 특성들과 메소드들, 이벤트들을 제공하는 지를 찾는 과정이다. 내성은 두가지 방법으로 수행된다:

- **특정 명명 패턴들을 따르는 클래스들과 메소드들을 검색하는 것에 의해**
- **클래스의 BeanInfo를 조회하는 것에 의해**

일반적으로, 내성은 빈 환경을 위해 예약된 활동이다. 빈 환경은 빈들에 대해 배우기위해 내성을 사용한다. 그러나 빈들은 스스로 내성을 수행할 필요가 없다. 그러나 하나의 빈이 다른 빈들을 분석하기 위해 내성을 사용할 필요가 있는 경우는 스스로 내성을 수행하는 경우이다. 좋은 예로는 여러분이 빌더 도구에서 하나의 폼상에 두개의 빈을 밀접하게 연결하길 원할 경우를 들 수 있다. 가령, 스핀 빈을 고려해 보자. 두개의 버튼을 갖는 작은 컨트롤 항목, 값을 증가시키거나 감소시킨다.(그림8-26 참조)

그림 8-26 : 스핀 빈

스핀 빈은 그 자체로 유용한 것은 아니다. 다른 빈들과 연결될 필요가 있다. 가령, 스핀 빈은 정수 텍스트 빈과 연결될 수 있다. 사용자가 스핀 빈의 버튼중의 하나를 클릭할 때마다, 정수 값은 증가하거나 감소한다. 우리는 연결된 빈을 스핀 빈의 ***버디****(Buddy)*라 부를 것이다. 버디는 IntTextBean일 필요가 없다. 정수 속성을 갖는 임의의 다른 빈 일 수 있다.

여러분은 버디를 부착하기 위해 스핀 빈의 커스터마이져를 사용한다.(그림 8-27).

그림8-27 : SpinBean의 커스터마이져들

다음은 이를 실행하는 방법이다.

1. 폼상에 SpinBean과 IntTextBean을 추가한다.
2. 메뉴상에서 Edit->Customize를 선택하여 스핀 빈의 커스터마이저를 팝업한다.
3. 버디 목록에서 IntTextBean을 선택한다.
4. 속성 목록에 있는 모든 int 속성들이 어떻게 자동적으로 채워지는지 살펴본다(그림 8-27).
5. value를 선택하고 Set Buddy를 클릭한다.
6. “Done”을 클릭한다.
7. 그런 다음, “+”와 “-“를 클릭하고 정수 텍스트 필드 값이 증가하고 감소하는 것을 살펴본다(그림 8-28 참조).

그림 8-28 : IntTextBean 버디와 연결되어 있는 SpinBean

이것은 쉬워보이지만, 커스터마이징을 구현하는 데에는 2가지 도전이 있다.

- int 형의 값들을 갖는 빈의 모든 속성들을 발견하는 방법은?
- 만일 여러분이 이를 오직 실시간에 알게된다면 프로그램에서 속성을 가져오고 설정하는 방법은?

우리는 이러한 두가지 문제를 해결하기 위해 내성(즉, 내성 API)을 사용한다. 빈을 속성을 분석하기 위해, 먼저 Introspector 클래스의 getBeanInfo 정적 메소드를 호출하는 것에 의해서 빈 정보를 얻는다.

```
BeanInfo info
        = Introspector.getBeanInfo(buddy.getClass());
```

일단 빈 정보를 얻은 다음, 속성 설명자들의 배열을 얻을 수 있다:

```
PropertyDescriptor[] props = info.getPropertyDescriptors();
```

스핀 빈 커스터마이저에서, 다음 단계는 해당 배열을에 대해 루프를 도는 것이다. Int형의 모든 속성들을 선택하고 그들의 이름을 리스트 컴포넌트에 추가한다.

```
for(int i=0; i < props.length; i++)
{       Class propertyType = props[i].getPropertyType();
        if(int.class.equals(propertyType))
        {       String name = props[i].getName();
                propModel.addElement(name);
        }
}
```

이 코드는 여러분이 빈의 속성들을 어떻게 발견할 수 있는 가를 보여준다.

다음, 우리는 사용자가 선택한 속성들을 얻고 설정할 수 있는 기능이 필요하다. 우리는 getReadMethod와 getWriteMethod를 호출하여 get과 set 메소드를 얻을 수 있다.

```
Method getMethod = prop.getReadMethod();
Method getMethod = prop.getWriteMethod();
```

(왜 getReadMethod를 호출하는 것일까? 아마 getGetMethod이란 표기가 다소 이상해 보이기 때문일 것이다.)

이제, 우리는 값을 얻고 증가하고 설정하는 메소드들을 호출한다. 이 과정은 리플레션 API를 다시 사용한다 – 1권의 5장 참조. 우리는 int 값주변에 Integer 포장자를 사용해야 한다.

```
int value = ( ( Integer)getMethod.invoke(buddy, null)).intValue();
value += increment;
setMethod.invoke(buddy, new Object[] { new Integer(value) } );
```

만일 버디가 getValue**와** setValue **메소드를 가지고 있길 요구한다면 리플렉션을 피할 수 있을까**? **불가능하다**.

```
int value = buddy.getValue();
```

컴파일러가 buddt**가** getValue **메소드를 가지고 있는 타입의 객체라는 것을 알 때 만 오직 호출할 수 있다**. **그러나** buddy**는 모든 타입일 수 있다** – **빈들에 대한 타입 계층도는 존재하지 않는다**. **하나의 빈이 임의의 다른 빈과 연결되어 있을 때마다**, **내성을 사용할 필요가 있다**.

동등 빈 찾기

앞절에서, **여러분은 스핀 빈 버튼들이 버디 컴포넌트의 값을 변경하는 방법을 보았다**. **그러나**, **관련없는 다른 이슈가 존재한다-어떻게 스핀 빈 커스터마이져가 사용자에게 모든 가능한 버디들을 제공할 까**?

폼상에 모든 빈들을 나열하는 것은 여러분이 생각하는 것보다 쉽지 않다. **예를 들면**, **빈박스에서 여러분은 빈의 모든 형제**(sibling)**를 얻기위해 단순히 호출할 수 없다**.

```
Component[] siblings = getParent().getComponents()
```

여러분이 이렇게 할 수 없는 이유는 빈박스가 패널안에서 모든 빈을 둘러싸고 있기 때문이다.(**우리는 빈을 선택하기 위해 클릭한 마우스를 감지하고 선택된 빈 주변에 윤곽을 그려야 한다고 추축한다**. **그래서 빈박스에서 우리는 다음처럼 작성해야 한다**.

```
Component[] siblings = getParent().getParent().getParent().getComponents()
```

그러나, **이 해결책은 다른 빌더 환경에서 작동한다고 보장하지 못한다** – **왜냐하면 이러한 환경들은 이러한 모든 별도의 패널들이 필요하지 않을 만큼 충분히 똑똑하기 때문이다**.

자바2 **플랫폼에서**, **우리는 이 문제를 해결할 수 있다**. **자바 빈 구조는 이제** ***빈 컨텍스***(*Bean Context*)**의 개념을 지원한다**. **빈 컨텍스는 빈들과 빈 컨테이너사이의 논리적인 포함 관계를 표현할 수 있다**. **여러분이 보게 될 것처럼 물리적인 포함 관계와는 아주 다르다**.

빈 컨텍스는 빈들과 서비스들, 그리고 다른 빈 컨텍스들을 유지할 수 있다. AWT 컨테이너처럼 컨포넌트들과 다른 컨테이너들을 유지할 수 있다 (그림 8-29 참조).

그림 8-29 : 빈 컨텍스

현재, 중첩된 빈 컨텍스들이 일반적인 것은 아니다. 최선은 , 여러분이 빈들이 단일 빈 컨텍스, 빌더 환경내에 살아있도록 기대할 수 있다. 가령, BDK1.1에서 빈박스는 빈 컨텍스이다. 빈박스는 빈들이 로깅 메시지를 출력하기 위해 사용할 수 있는 *메시지 추적 서비스*를 제공한다. 다른 빈 컨텍스는 다른 서비스들을 제공할 수 잇다. 다음 절에서, 우리는 여러분에게 여러분의 빈들이 메시지 추적 서비스를 사용할 수 있는 방법을 보여줄 것이다.

노트 : BDK의 향후 버전은 인포버스(InfoBus) 서비스를 제공할 것이다. 인포버스 기술에 대한 보다 자세한 정보는 다음 URL을 참조하라.

http://java.sun.com/beans/infobus/index.html

주의 : 현재 이용가능한 모든 빌더 환경들이 그들의 빈들을 빈 컨텍스내에 위치시킬 수 있는 것은 아니다. 비록 여러분의 환경이 빈 컨텍스를 사용하더라도, 메시지 추적 서비스를 제공하지 않을 수 도 있다. 그러므로, 예제 프로그램은 여러분이 선호하는 환경하에서 작동하지 않을 수 있다.

빈 컨텍스 주변과 통신하기 위해, 빈은 BeanContextChild 인터페이스를 구현할 수 있다. 해당 인터페이스는 6개의 메소드들 가지고 있다:

```
void setBeanContext(BeanContext bc)
BeanContext getBeanContext()
void addPropertyChangeListener(Strong name, PropertyChangeListener listener)
void removePropertyChangeListener(Strong name, PropertyChangeListener listener)
void addVetoableChangeListener(Strong name, VetoableChangeListenerlistener)
void removeVetoableChangeListener(Strong name, VetoableChangeListenerlistener)
```

빈 컨텍스는 여러분의 빈을 관리하기 시작할 때 setBeanContext 메소드를 호출한다. 여러분은 BeanContext 전달인자에 값을 저장하고 이를 빈 컨텍스에 접근할 필요가 있을 때마다 사용할 수 있다.

리스너들의 변경을 구현하는 것이 단조로운 일이기 때문에, 여러분들을 위해 이러한 메소드들을 구현한 BeanContextChildSupport 클래스가 있다. 그러나 만일 여러분이 이 편리한 클래스를 사용하길 원할 것인가는 기술적인 이슈이다. 왜냐하면 여러분의 빈은 동시에 컴포넌트 클래스와 BeanContextChildSupport 클래스를 확장할 수 없기때문이다. 여러분은 프록시 방법을 사용한다. 이들 과정은 다음과 같다.

1. BeanContextProxy 인터페이스를 구현한다.
2. BeanContextChildSupport 타입의 인스턴스 변수를 구성한다.

```
private BeanContextChildSupport childSupport
        = new BeanContextChildSupport();
```

3. BeanContextProxy 인터페이스의 get BeanContextProxy 메소드내에 인스턴트 변수를 반환한다.

```
public BeanContextChild getBeanContextProxy()
{       return childSupport;
}
```

   빈 컨텍스는 프록시 객체를 추출하기 위해 이 메소드를 호출한다. 그런 다음 빈 그 자체보다 반환된 객체의 BeanContextChild 메소드들을 호출한다.
4. 여러분이 현재 빈 컨텍스를 알길 원할 때마다 childSupport 객체의 getBeanContext 메소드를 호출한다.

```
BeanContext context = childSupport.getBeanContext();
```

그림 8-30 이들 클래스들 사이의 관계를 보여준다.

그림 8-30 : 빈 컨텍스 클래스들 사이의 관계

일단 여러분이 빈 컨텍스를 갖으면, 여러분은 여기에 포함되어 있는 모든 빈들을 나열할 수 있다. BeanContext 클래스는 Collection 인터페이스를 구현한다. 그러므로, 여러분은 다음처럼 손쉽게 빈들을 나열할 수 있다.

```
Iterator iter = beanContext.iterator();
while(iter.hasNext() )
{       Object buddy = iter.next();
        do something with buddy
}
```

사용자 인터페이스 관점에서, 가능한 버디들의 타입들을 나열하는 우리의 접근 방법은 여전히 만족스러운 것은 아니다. 만일 폼에 같은 클래스의 형제(sibling)가 하나 이상 존재할 때, 사용자는 이들이 떨어져 있다고 말할 수 없다. 만일 우리가 물리적으로 스핀 빈을 해당 버디에 가깝게 이동하거나 선택된 버디주변에 박스를 그린다면 좋을 것이다. 그러나 BeanContext는 위치 정보를 제공하지 않는다.(버디에게 해당 위치를 묻는 것은 소용없는 일이다. 오직 닫혀진 패널에 대한 위치만을 알고 있다.) 아마 자바 빈 명세의 향후 버전은 이와 같은 조작을 가능하게 해 줄 것이다.

빈 컨텍스트 서비스 사용하기

이 절에서, 우리는 여러분의 빈들이 빈 컨텍스에서 제공하는 서비스들에 접근하는 방법을 보여주길 원한다. 여러분은 여러분이 원하는 서비스를 제공하는 클래스를 알 필요가 있다. 가령, 빈 박스가 제공하는 메시지 추적 서비스는 sunw.demo.methodtracer.MethodTracer 클래스에서 구현된다. 먼저, 여러분은 빈 컨텍스가 BeanContextServices 인터페이스를 구현했는지 확인할 필요가 있다 – 모든 빈 컨텍스가 그러한 것은 아니다.

```
if(beanContext implements BeanContextServices)
{       BeanContextServices services
                = (BeanContextServices)beanContext;
        …
}
```

그런 다음, 원하는 서비스를 제공하는 지를 빈 컨텍스에 요청한다.

```
traceClass = Class.forName
        (“sunw.demo.methodtracer.MethodTracer”);
if(services.hasService(tracerClass) )
{       …
}
```

만일 여러분이 빈 박스가 아닌 다른 환경에서 해당 빈 컨텍스를 수행한다면, hasService 호출은 flase를 반환할 것이다.

마지막으로, 여러분은 서비스를 실행하는 객체를 얻을 필요가 있다. 여러분은 5개의 전달인자를 갖는 getService 메소드를 호출한다.

- ChildBeanContext 인터페이스를 구현한 객체; 일반적으로 BeanContextChildSupport 프록시 객체
- 요청하는 객체
- 여러분이 얻길 원하는 서비스의 클래스
- 만일 서비스가 요구하면 올바른 서비스를 선택하기 위한 보조 객체를 제공한다, 그렇지 않으면 null을 반환한다.
- 서비스 이용이 중단됐을 때 호출되는 BeanContextServiceRevokedListener 객체

다음은 기본적인 호출이다.

```
BeanContextServiceRevokedListener revokedListener =
        new BeanContextServiceRevokedListener()
{       public void serviceRevoked(BeanContextServiceRevokedEvent event)
        {       tracer = null;
        }
};
tracer = services.getService(childSupport, this, tracerClass, null, revokedListener);
```

여러분은 기본적으로 여러분의 프로그램을 컴파일할 때 이용가능한 서비스 클래스를 갖지않는다는 것을 기억하라. 가령, MethodTracer 클래스를 위한 클래스 파일은 표준 실행 라이브러리의 일부가 아니다. 그리고 컴파일러는 비록 여러분이 sunw.demo.methodtracer 패키지를 임포트하더라도 발견하지 못할 것이다.

이러한 이유 때문에, 우리는 서비스 메소드들을 호출하기 위해 리플렉션을 사용한다. MethodTracer 클래스의 logText 메소드를 호출하는 다음 구문은 메소드 추적 윈도우내에 메시지를 출력한다.

```
if(tracer != null)
{       String text = "spin: value= " + value + " increment=" + increment;
        Method logText = tracerClass.getMethod("logText",
                new Class[] { String.class} )
```

```
                logText.invoke(tracer, new Object[] {text} );
        }
```

물론, 빈 컨텍스가 서비스를 지원하지 않는다면 메소드를 호출하지 않는다. 만일 지원한다면 우리는 리플렉션 메커니즘을 사용하여 이를 호출하기 위해 invoke 메소드를 사용한다.

여러분이 해당 예제를 시도할 때, 메시지 추적 윈도우를 지켜 볼 것이다. 이 윈도우는 스핀 버튼들중의 하나를 누를 때 생성되는 메시지를 포함한다.(그림 8-31 참조)

그림 8-31 : 메시지 추적 윈도우

예제 8-17에서부터 8-19까지 에는 스핀 빈의 완전한 코드가 포함되어 있다.

예제 8-17 : SpinBean.java

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.beans.beancontext.*;
import java.lang.reflect.*;
import java.io.*;
import java.util.*;
import javax.swing.*;

public class SpinBean extends JPanel
   implements Serializable, BeanContextProxy
{   public SpinBean()
    {   setLayout(new GridLayout(2, 1));
       JButton plusButton = new JButton("+");
       JButton minusButton = new JButton("-");
        add(plusButton);
        add(minusButton);
        plusButton.addActionListener(
           new ActionListener()
            {   public void actionPerformed(ActionEvent evt)
                {   spin(1);
```

```
            }
        });
    minusButton.addActionListener(
        new ActionListener()
        {   public void actionPerformed(ActionEvent evt)
            {   spin(-1);
            }
        });

    childSupport =
        new BeanContextChildSupport()
        {   public void setBeanContext(BeanContext context)
                throws PropertyVetoException
            {   super.setBeanContext(context);
                setTracer(context);
            }
        };
}

public BeanContextChild getBeanContextProxy()
{   return childSupport;
}

public void setBuddy(Component b, PropertyDescriptor p)
{   buddy = b;
    prop = p;
}

public void spin(int increment)
{   if (buddy == null) return;
    if (prop == null) return;
    Method readMethod = prop.getReadMethod();
    Method writeMethod = prop.getWriteMethod();
    try
    {   int value = ((Integer)readMethod.invoke(buddy,
            null)).intValue();
```

```
        if (tracer != null)
        {   String text = "spin: value=" + value
                + " increment=" + increment;
            Method logText = tracerClass.getMethod("logText",
                new Class[] { String.class });
             logText.invoke(tracer, new Object[] { text });
          }

         value += increment;
          writeMethod.invoke(buddy,
            new Object[] { new Integer(value) });
      }
     catch(Exception e)
      {
      }
  }

public void setTracer(BeanContext context)
 {   try
     {   BeanContextServices services
            = (BeanContextServices)context;
         tracerClass = Class.forName
              ("sunw.demo.methodtracer.MethodTracer");
          if (services.hasService(tracerClass))
          {   BeanContextServiceRevokedListener revokedListener =
                  new BeanContextServiceRevokedListener()
                  {   public void serviceRevoked
                          (BeanContextServiceRevokedEvent event)
                      {   tracer = null;
                       }
                   };
              tracer = services.getService(childSupport, this,
                  tracerClass, null, revokedListener);
          }
      }
```

```
        catch (Exception exception)
        {   tracer = null;
         }
    }

   public Dimension getPreferredSize()
   {   return new Dimension(MINSIZE, MINSIZE);
    }

  private static final int MINSIZE = 20;
   private Component buddy;
   private PropertyDescriptor prop;
   private Object tracer;
   private Class tracerClass;
   private BeanContextChildSupport childSupport;
}
```

예제 8-18 : SpinBeanCustomizer.java

```
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.beans.beancontext.*;
import java.io.*;
import java.text.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

public class SpinBeanCustomizer extends JPanel
    implements Customizer
{   public SpinBeanCustomizer()
    {   setLayout(new GridBagLayout());
       GridBagConstraints gbc = new GridBagConstraints();
        gbc.weightx = 0;
        gbc.weighty = 100;
```

```
    gbc.fill = GridBagConstraints.NONE;
   gbc.anchor = GridBagConstraints.EAST;
  add(new JLabel("Buddy"), gbc, 0, 0, 1, 1);
  add(new JLabel("Property"), gbc, 0, 1, 1, 1);
   gbc.weightx = 100;
   gbc.anchor = GridBagConstraints.WEST;
    gbc.fill = GridBagConstraints.BOTH;
   buddyModel = new DefaultListModel();
   propModel = new DefaultListModel();
   buddyList = new JList(buddyModel);
   propList = new JList(propModel);
  add(new JScrollPane(buddyList), gbc, 1, 0, 1, 1);
   add(new JScrollPane(propList), gbc, 1, 1, 1, 1);
  JButton setButton = new JButton("Set Buddy");
  JPanel p = new JPanel();
    p.add(setButton);
  add(p, gbc, 0, 2, 2, 1);

    buddyList.addListSelectionListener(
        new ListSelectionListener()
        {   public void valueChanged(ListSelectionEvent event)
            {   findBuddyMethods();
            }
         });

    setButton.addActionListener(
        new ActionListener()
        {   public void actionPerformed(ActionEvent event)
            {   int buddyIndex = buddyList.getSelectedIndex();
              if (buddyIndex < 0) return;
               int propIndex = propList.getSelectedIndex();
              if (propIndex < 0) return;
                bean.setBuddy(buddies[buddyIndex], props[propIndex]);
            }
         });
}
```

```
public void add(Component c, GridBagConstraints gbc,
   int x, int y, int w, int h)
{   gbc.gridx = x;
    gbc.gridy = y;
    gbc.gridwidth = w;
    gbc.gridheight = h;
    add(c, gbc);
 }

public void findBuddyMethods()
{   int buddyIndex = buddyList.getSelectedIndex();
   if (buddyIndex < 0) return;
   Component buddy = buddies[buddyIndex];
    propModel.removeAllElements();
    try
    {   BeanInfo info
            = Introspector.getBeanInfo(buddy.getClass());
       props = info.getPropertyDescriptors();
       int j = 0;
      for (int i = 0; i < props.length; i++)
       {   Class propertyType = props[i].getPropertyType();
           if (int.class.equals(propertyType))
           {   String name = props[i].getName();
                propModel.addElement(name);
               props[j++] = props[i];
            }
         }
     }
    catch(IntrospectionException e){}
 }

public Dimension getPreferredSize()
{   return new Dimension(300, 200);
 }
```

```
public void setObject(Object obj)
{   bean = (SpinBean)obj;
    BeanContext context = bean.getBeanContextProxy().getBeanContext();
    buddies = new Component[context.size()];
     buddyModel.removeAllElements();
    Iterator iter = context.iterator();
    int i = 0;
     while (iter.hasNext())
    {   Object buddy = iter.next();
        if (buddy instanceof Component)
         {   buddies[i] = (Component)buddy;
             String className = buddies[i].getClass().getName();
              buddyModel.addElement(className);
              i++;
          }
     }
}

public void addPropertyChangeListener
    (PropertyChangeListener l)
{   support.addPropertyChangeListener(l);
}

 public void removePropertyChangeListener
    (PropertyChangeListener l)
{   support.removePropertyChangeListener(l);
}

private SpinBean bean;
private PropertyChangeSupport support
   = new PropertyChangeSupport(this);
private JList buddyList;
private JList propList;
private DefaultListModel buddyModel;
private DefaultListModel propModel;
private PropertyDescriptor[] props;
```

```
   private Component[] buddies;
}
```

예제 8-19 : SpinBeanBeanInfo.java

```
import java.awt.*;
import java.beans.*;

public class SpinBeanBeanInfo extends SimpleBeanInfo
{  public BeanDescriptor getBeanDescriptor()
   {  return new BeanDescriptor(SpinBean.class,
            SpinBeanCustomizer.class);
    }
}
```

java.beans.Introspector

- string decapitalize(String name)
  문자열을 자바 플랫폼 명명 규칙으로 변환한다. **예를 들면**, SillyMethod**는** sillyMethod**가 된다**.(**두개의 대문자가 연속될 때는 아무런 변화도 발생하지 않는다**.)
- BeanInfo getBeanInfo(Class beanClass)
  빈과 연관된 BeanInfo **클래스를 얻거나 이 장의 앞에서 설명한 명명 규칙을 사용하여 이 클래스를 생성한다**. **만일 내성이 실패하면** IntrospectionException**을 발생한다**.

java.beans.beancontext.BeanContextChild

- void setBeanContext(BeanContext bc)
  빈 컨텍스가 해당 빈을 자식으로 추가할 때 호출한다. **모든 거부가능한 리스너들에게** "beanContext" **속성 이름을 갖는 다섯개의 속성 변경 이벤트를 발생해야 한다**.
- BeanContext getBeanContext()
  해당 빈에 대한 현재의 빈 컨텍스를 반환한다.
- void addPropetyChangeListener(String name, PropetyChangeListener listener)
  명명된 속성에 대해 리스너를 추가한다.
- void removePropetyChangeListener(String name, PropetyChangeListener listener)
  명명된 속성에 대해 리스너를 제거한다.
- void addVetoableChangeListener(String name, VetoableChangeListener listener)
  명명된 속성에 대해 거부가능한 리스너를 추가한다.

- void removeVetoableChangeListener(String name, VetoableChangeListener listener)
  명명된 속성에 대해 거부가능한 리스너를 제거한다.

java.beans.beancontext.BeanContextChildSupport

- BeanContext getBeanContext()
  해당 빈에 대한 현재의 빈 컨텍스를 반환한다.

javax.beans.beancontext.BeanContextProxy

- BeanContextChild getBeanContextProxy()
  BeanContextChild **메소드를 다루는 프록시 객체를 반환한다**.

javax.beans.beancontext.BeanContextServices

- boolean hasService(Class cl)
- object getService(BeanContextChild child, Object requestor, Class cl, Object selector, BeanContextServiceRevokedListener listener)
  빈 컨텍스 서비스를 수행하는 객체를 가져온다.

전달인자:	child	**빈 컨텍스의 자식으로서 연결되어 있는 빈 \ 컨텍스 자식 객체**
	requestor	**서비스를 요청하는 객체**
	cl	**서비스 클래스**
	selector	**서비스를 위치시키거나 초기화하는 옵션 객체**
	listener	**서비스를 더 이상 이용할 수 없을 때 알려줄 리스너**

javax.beans.beancontext.BeanContextServiceRevokedListener

- void serviceRevoked(BeanContextServiceRevokedEvent event)
  서비스가 철회됐을 때 호출된다.

javax.beans.beancontext.BeanContextServiceRevokedEvent

- Class getServiceClass()
  철회된 서비스를 실행하는 클래스를 얻는다.
- boolean isServiceClass(Class cl)
  주어진 클래스 객체가 철회된 서비스를 실행하는 클래스를 설명하는지 검사한다.